Pseudorandom Functions with Weak Programming Privacy and Applications to Private Information Retrieval

Ashrujit Ghoshal[1], Mingxun Zhou[1], Elaine Shi[1], and Bo Peng[*1,2]

[1]Carnegie Mellon University
[2]Peking University

Abstract

Although privately programmable pseudorandom functions (PPPRFs) are known to have numerous applications, so far, the only known constructions rely on Learning with Error (LWE) or indistinguishability obfuscation. We show how to construct a relaxed PPPRF with only one-way functions (OWF). The resulting PPPRF satisfies 1/poly security and works for polynomially sized input domains. Using the resulting PPPRF, we can get new results for preprocessing Private Information Retrieval (PIR) that improve the state of the art. Specifically, we show that relying only on OWF, we can get a 2-server preprocessing PIR with polylogarithmic bandwidth while consuming $\widetilde{O}_\lambda(N^{\frac{1}{2}+\epsilon})$ client space and $N^{1+\epsilon}$ server space for an arbitrarily small constant $\epsilon \in (0,1)$. In the 1-server setting, we get a preprocessing PIR from OWF that achieves polylogarithmic *online* bandwidth and $\widetilde{O}_\lambda(N^{\frac{1}{2}+\epsilon})$ *offline* bandwidth, while preserving the same client and server space as before. Our result, in combination with the lower bound of Ishai, Shi, and Wichs (CRYPTO'24), establishes a tight understanding of the bandwidth and client space tradeoff for 1-server preprocessing PIR from Minicrypt assumptions. Interestingly, we are also the first to show non-trivial ways to combine client-side and server-side preprocessing to get improved results for PIR.

*Randomized Author Ordering. Bo Peng contributed to this work during a visit at CMU. Email: {aghoshal,mingxunz,rshi}@andrew.cmu.edu, bo.peng@stu.pku.edu.cn.

Contents

1 Introduction

A programmable pseudorandom function (PRF) [BW13, BGI14] is a PRF with an additional programming feature: given a normal key, one can generate a programmed key such that the evaluation of the PRF at some specified location x is forced to a particular value v, whereas evaluations at all other locations are unchanged. A programmable PRF satisfies *programming privacy* [BLW17, BKM17, CC17, PS18], if the programmed key does not leak the point x at which the programming took place, as long as the programmed value v is randomly chosen. Such PRFs are called Privately Programmable PRF (PPPRF) [BLW17, BKM17, CC17, PS18]. It is well-known that if we do not need programming privacy, we can construct programmable PRFs from one-way functions [BW13, BGI14], relying on the famous tree-based construction of Goldreich, Goldwasser, and Micali [GGM86], and additionally specifying the point x and the value v in the programmed key. Further, the resulting construction is simple and enjoys concrete efficiency. By contrast, if we insist on programming privacy, to the best of our knowledge, PPPRF constructions are only known from lattice assumptions such as Learning with Errors (LWE) [CC17, BTVW17, PS18] or from indistinguishability obfuscation (iO) [BLW17]. Further, known constructions are complex, and rely on layers of fully homomorphic encryption.

The programming privacy feature, however, turns out to be crucial in many applications. A notable example is the recent line of work on client-side preprocessing Private Information Retrieval (PIR). We now give some background on PIR and explain how PPPRFs aids the construction of modern PIR schemes.

PIR allows a client to request entries from large, public database stored by one or more server(s), without leaking its query to any individual server [CGKS95]. Because PIR promises many useful applications such as private DNS [pri], private blocklist [KCG21], and private web search [HDCG+23], the primitive has been extensively studied for several decades. Classical PIR schemes do not employ preprocessing and require that the server store only the original database. It is well-known that in the classical model, any PIR scheme must suffer from linear cost per query [BIM00]: informally, if there is some location that the server need not visit during a query, it leaks that the client is not interested in that location. Fortunately, more recent works showed that we can overcome this linear computation barrier through either *server-side* preprocessing [BIM00, WY05] or *client-side* preprocessing [CK20]. In server-side preprocessing, the server encodes the database and stores the encoded version. In client-side preprocessing, each client runs some protocol with the server once upfront to subscribe to the private query service. At the end of the preprocessing protocol, the client stores some hint that depends on the database. The hint will facilitate the client's future queries, and we want the hint to be asymptotically smaller than the original database. It is desirable if after the one-time preprocessing, we can support an unbounded number of queries — in other words, the total number of queries is unknown at preprocessing time.

A line of recent works [CK20, SACM21, GZS24, ZPSZ24, CHK22, ZLTS23] have shown a close connection between PPPRFs and client-side preprocessing PIR with sublinear computation per query. Notably, Shi et al. [SACM21] showed that assuming the existence of a PPPRF, we can get a 2-server client-preprocessing PIR with polylogarithmic bandwidth and $\widetilde{O}_\lambda(\sqrt{N})$ client and server computation per query, assuming $\widetilde{O}_\lambda(\sqrt{N})$ client space, where N denotes the database size, and $\widetilde{O}_\lambda(\cdot)$ hides polylogarithmic terms and a polynomial dependence on the security parameter λ. Their approach, in turn inspired by Corrigan-Gibbs and Kogan [CK20], is to use a Privately Puncturable PRF (a slightly weaker variant of Privately Programmable PRF also known only from LWE or iO) to construct a privately puncturable pseudorandom set (PPPS), and then use the latter as a stepping stone towards getting PIR. Subsequent works showed that relying on PPPRF and fully homomorphic encryption (FHE), we can extend Shi et al.'s result to the single-server

setting [ZLTS23, LP22] while preserving the same asymptotics*. Since the existence of PPPRF is known from LWE or iO [BLW17, BKM17, CC17, PS18], these works [SACM21, CHK22, ZLTS23, LP22] establish the theoretical feasibility of achieving preprocessing PIR with the aforementioned costs, assuming either LWE or iO.

Of course, we are not satisfied with just theoretical feasibility, which prompted the community's effort in search of a preproccessing PIR scheme with better concrete efficiency [LP23, ZPSZ24, MIR23, HPPY24, GZS24]. As a stepping stone towards concrete efficiency, recent works [ZPSZ24, MIR23, HPPY24, GZS24] asked whether we can get efficient preprocessing PIR from minimal assumptions [ZPSZ24, MIR23, HPPY24]. Specifically, rather than relying on LWE or iO, *can we get efficient preprocessing PIR with only symmetric primitives such as one-way functions (OWFs)*? This line of work culminated in the result of Ghoshal et al. [GZS24], who showed that assuming only OWF, we can get $O_\lambda(N^{1/4})$ bandwidth per query in the 2-server setting, while all other costs remain the same as Shi et al. [SACM21]. Although non-trivial, this result is not completely satisfying since we cannot match the polylogarithmic bandwidth cost of Shi et al. [SACM21]. For the 1-server setting, Ghoshal et al. [GZS24] showed that assuming only OWF, we can get $O_\lambda(N^{1/4})$ *online* bandwidth and the same $O_\lambda(N^{1/2})$ *offline* bandwidth as before, while all other costs remain the same. The subsequent work of Ishai et al. [ISW24] completed the picture by showing that in the 1-server setting, the $O_\lambda(N^{1/2})$ offline bandwidth cannot be improved under $O(\sqrt{N})$ client space, unless we make cryptographic assumptions that imply public-key primitives. At a very high level, Ghoshal et al.'s approach side-steps the issue of lacking a suitable PPPRF from OWF, by using OWF to directly construct a privately puncturable pseudorandom set that provides only list decoding rather than unique decoding. However, the drawback is that the list decoding overhead translates directly to the scheme's bandwidth cost, which leaves us seemingly stuck at $O_\lambda(N^{1/4})$ (online) bandwidth with the techniques.

Thus, the state of the art of preprocessing PIR begs the following questions:

- *In the 2-server setting, can we match the asymptotics of Shi et al. [SACM21] but relying only on OWF?*
- *In the 1-server setting, although there is a barrier for the offline bandwidth, can we further improve the online bandwidth while preserving other costs?*

Now, if we can get an efficient PPPRF from OWF, we would be able to affirmatively answer the above questions. Unfortunately, the feasibility/infeasibility of constructing PPPRF from OWF remains open and progress towards either upper bound or (blackbox) separation would be very exciting. In this paper, we ask a more relaxed form of the question:

- *Can we construct slightly relaxed versions of PPPRFs from OWF that are nonetheless sufficient for constructing efficient preprocessing PIR?*

1.1 Our Results and Contributions

We consider a relaxed form PPPRF. First, we restrict ourselves to PPPRFs where the input domain is polynomial in size. Second, instead of aiming for full security, we will allow $1/\mathsf{poly}$ security failure. We show that with these relaxations, it is indeed possible to construct a reasonably efficient PPPRF from OWF. Moreover, we show that the relaxed PPPRF is indeed sufficient for constructing

*However, these schemes [ZLTS23, LP22], which partly build on top of the ideas of Corrigan-Gibbs et al. [CHK22], suffer from the following caveat: they require that the server allocate $\Omega_\lambda(N)$ space per client during the preprocessing phase. In particular, in the unbounded query model, the preprocessing of the next batch of queries is piggybacked on the current batch of queries, meaning that the server needs $\Omega_\lambda(N)$ per-client space constantly.

Table 1: **Comparison: 1-server PIR with sublinear computation.** N is the database size, m denotes the number of clients, and $\epsilon \in (0,1)$ is an arbitrarily small positive constant. Note*: Our bandwidth is $(\epsilon\lambda \log N)^{O(1/\epsilon)}$.

Scheme	Assumpt.	Compute	Communication		Space	
			online	offline	client	server
[CHK22]	LWE	$\widetilde{O}_\lambda(\sqrt{N})$	$\widetilde{O}_\lambda(\sqrt{N})$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{N})$	$\widetilde{O}_\lambda(m \cdot N)$
[ZLTS23, LP22]	LWE	$\widetilde{O}_\lambda(\sqrt{N})$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{N})$	$\widetilde{O}_\lambda(m \cdot N)$
[LMW23]	Ring-LWE	$O_\lambda((\log N)^{O(\frac{1}{\epsilon})})$	$\widetilde{O}_\lambda(1)$	0	0	$O_\lambda(N^{1+\epsilon})$
[ZPSZ24, MIR23]	OWF	$\widetilde{O}_\lambda(\sqrt{N})$	$O_\lambda(\sqrt{N})$	$O_\lambda(\sqrt{N})$	$\widetilde{O}_\lambda(\sqrt{N})$	$O(N)$
[GZS24]	OWF	$\widetilde{O}_\lambda(\sqrt{N})$	$O_\lambda(N^{1/4})$	$O_\lambda(\sqrt{N})$	$\widetilde{O}_\lambda(\sqrt{N})$	$O(N)$
Ours	OWF	$\widetilde{O}_\lambda(N^{\frac{1}{2}+\epsilon})$	$\widetilde{O}_\lambda(1)$ *	$O_\lambda(N^{\frac{1}{2}+\epsilon})$	$\widetilde{O}_\lambda(N^{\frac{1}{2}+\epsilon})$	$N^{1+\epsilon}$

efficient preprocessing PIR, which allows us to affirmatively answer the above questions that are left open by the recent PIR literature. We now elaborate on our results.

We first present our main result on weakly secure, small-domain PPPRF.

Theorem 1.1 (1/poly-secure, small-domain PPPRF)**.** *Assume the existence of OWF. Then, for any arbitrarily small $\epsilon \in (0,1)$, there exists a PPPRF with δ-programming privacy that maps ℓ_{in}-bit inputs to ℓ_{out}-bit outputs, with the following performance where $n = 2^{\ell_{\text{in}}}$ denotes the size of the input domain:*

- *normal and programmed key length* $= (\epsilon\lambda \log n)^{O(1/\epsilon)} \cdot \ell_{\text{out}}/\delta^2$
- *evaluation and programming time* $= (\epsilon\lambda \log n)^{O(1/\epsilon)} \cdot n^\epsilon \cdot \ell_{\text{out}}^3/\delta^4$.

Specifically, for $\delta = 1/\mathsf{poly}(\lambda)$, the key length is upper bounded by $\ell_{\text{out}} \cdot \mathsf{poly}(\lambda, \log n)$, and the evaluation and programming time is bounded by $\ell_{\text{out}}^3 \cdot n^\epsilon \cdot \mathsf{poly}(\lambda, \log n)$.

With Theorem 1.1, we prove the following results about preprocessing PIR.

Theorem 1.2 (2-server preprocessing PIR from OWF)**.** *Assume the existence of OWF. Then, for any arbitrarily small $\epsilon \in (0,1)$, there exists a 2-server preprocessing PIR for an N-bit database, with $(\epsilon\lambda \log N)^{O(1/\epsilon)}$ bandwidth and $N^{1/2+\epsilon}(\epsilon\lambda \log N)^{O(1/\epsilon)}$ server and client computation per query, requiring $N^{1/2+\epsilon}(\epsilon\lambda \log N)^{O(1/\epsilon)}$ client space and $N^{1+\epsilon} \log^{O(1/\epsilon)}(N, \lambda)$ server space.*

Simplifying the $(\epsilon\lambda \log N)^{O(1/\epsilon)}$ and $\log^{O(1/\epsilon)}(N, \lambda)$ factors to $\widetilde{O}_\lambda(1)$, and suppose that N is polynomially bounded in λ, effectively our 2-server preprocessing PIR scheme enjoys $\widetilde{O}_\lambda(1)$ bandwidth and $\widetilde{O}_\lambda(N^{\frac{1}{2}+\epsilon})$ client and server computation per query, with $\widetilde{O}_\lambda(N^{\frac{1}{2}+\epsilon})$ client space and $N^{1+\epsilon}$ server space.

Theorem 1.3 (1-server preprocessing PIR from OWF)**.** *Assume the existence of OWF. Then, for any arbitrarily small $\epsilon \in (0,1)$, there exists a 1-server preprocessing PIR for an N-bit database, with $(\epsilon\lambda \log N)^{O(1/\epsilon)}$ online bandwidth, and $N^{1/2+\epsilon}(\epsilon\lambda \log N)^{O(1/\epsilon)}$ offline bandwidth, server and client computation per query, requiring $N^{1/2+\epsilon}(\epsilon\lambda \log N)^{O(1/\epsilon)}$ client space and $N^{1+\epsilon} \log^{O(1/\epsilon)}(N, \lambda)$ server space.*

Again, simplifying the expressions, we effectively have $\widetilde{O}_\lambda(1)$ online bandwidth, $\widetilde{O}_\lambda(N^{\frac{1}{2}+\epsilon})$ offline bandwidth, client and server computation per query, while requiring $\widetilde{O}_\lambda(N^{\frac{1}{2}+\epsilon})$ client space and $N^{1+\epsilon}$ server space.

Table 2: **Comparison: 2-server PIR with sublinear computation.** The assumption PIR* means any assumption that implies classical single-server PIR with polylogarithmic bandwidth.

Scheme	Assumpt.	Compute	Communication	Space	
				client	server
[BIM00]	None	$O(N/\epsilon^2 \log^2 N)$	$O(N^{1/3})$	0	$N^{1+\epsilon}$
[WY05]	None	$O(N/\mathsf{poly}\log N)$	$O(N^{1/3})$	0	$\mathsf{poly}(N)$
[BIM00, WY05]	None	$O(N^{1/2+\epsilon})$	$O(N^{1/2+\epsilon})$	0	$\mathsf{poly}(N)$
[ISW24]	None	$\widetilde{O}(\sqrt{N})$ client $\widetilde{O}(N^{2/3})$ server	$O(N^{1/3})$	$\widetilde{O}(N^{2/3})$	$O(N)$
[SACM21]	LWE	$\widetilde{O}_\lambda(\sqrt{N})$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{N})$	$O(N)$
[LP23]	PIR*	$\widetilde{O}_\lambda(\sqrt{N})$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{N})$	$O(N)$
[ZPSZ24, MIR23]	OWF	$\widetilde{O}_\lambda(\sqrt{N})$	$O_\lambda(\sqrt{N})$	$O_\lambda(\sqrt{N})$	$O(N)$
[GZS24]	OWF	$\widetilde{O}_\lambda(\sqrt{N})$	$O_\lambda(N^{1/4})$	$\widetilde{O}_\lambda(\sqrt{N})$	$O(N)$
Ours	OWF	$\widetilde{O}_\lambda(N^{\frac{1}{2}+\epsilon})$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(N^{\frac{1}{2}+\epsilon})$	$N^{1+\epsilon}$

Clearly, the online bandwidth is optimal up to polylogarithmic factors. Further, restricted to OWF, the offline bandwidth and client space tradeoff is also optimal up to N^ϵ factors due to the lower bound results of Ishai, Shi, and Wichs [ISW24]. Therefore, *our result, in conjunction with Ishai et al. [ISW24], completes our understanding of 1-server preprocessing PIR from OWF* (particularly regarding the bandwidth and client space tradeoff).

Corrigan-Gibbs et al. [CHK22] showed that the product of the client space and server computation must be at least N, but this lower bound applies only when the server stores just the original database. In comparison, our scheme combines both client-side and server-side preprocessing, and the server actually stores an encoded version of the database — therefore, it is not clear whether Corrigan-Gibbs et al. [CHK22]'s lower bound applies to our setting.

The fact that our PIR scheme combines client- and server-side preprocessing is also interesting. Prior to our work, the literature on client-side preprocessing and server-side preprocessing relied on disjoint techniques and it was not clear how to meaningfully combine these techniques. In this sense, our paper can be viewed as *the first to show how to combine client-side and server-side preprocessing to get non-trivial results.*

2 Informal Technical Roadmap

2.1 Inefficient PPPRF from OWF

Inspired by the programmable distributed point function of Boyle et al. [BGIK22], we can construct an inefficient PPPRF as follows. Suppose that the input domain of the PPPRF is $\{1, \ldots, n-1\}$ and the output domain is a single bit. We consider randomly throwing m balls into n bins (where $m = n \cdot \mathsf{poly}(\lambda)$), and the parity of the load in bin x is the output of the PPPRF at x before programming. Now, to program the evaluation of the PPPRF at x, we can just remove a ball from bin x to flip the parity if the program bit is different from the original bit [†]. To actually implement the "randomly throwing balls" and "removing a ball" operation, we can directly use

[†]In the case that the program value is the same as the original value, we remove a ball from a dummy bin.

the classical GGM-based puncturable PRF construction. We sample a key sk of an underlying puncturable PRF $\mathsf{pPRF} : \{0,1\}^\lambda \times [m] \to [n]$ to pseudorandomly throw m balls into n bins. When we need to remove a ball from bin x, we find a random location r such that $\mathsf{pPRF}(\mathsf{sk}, r) = x$, and puncture the pPRF at r. Then, the resulting punctured key sk' only contains the information of the remaining $m-1$ balls. We can directly view this sk' as the programmed key of our PPPRF, and the evaluation of this programmed key at x is the parity of bin x when we consider throwing those $m-1$ balls into n bins.

This construction provides some level of privacy because when we puncture the underlying pPRF at the point r, the punctured key does not leak the pPRF's original evaluation outcome at r, so the adversary cannot easily tell which location has been programmed. On the other hand, the scheme has some leakage, because after puncturing the underlying pPRF at r, the joint bin load distribution get somewhat skewed, and the adversary can potentially glean information from this skewed distribution. From a careful analysis, one can show that the resulting PPPRF has privacy loss roughly $\sqrt{n/m}$. In other words, if we set $m = n \cdot \mathsf{poly}(\lambda)$, we can get $1/\mathsf{poly}'(\lambda)$-programming privacy.

Drawback: poor efficiency. Recall that we are using the m balls into n bins construction, and the outputs of the PPPRF are the parities of the bin loads. Even to evaluate at a single point of the PPPRF, one needs to actually enumerate all the m balls to count the parity. Jumping ahead, in our PIR construction later, for every query, the client will need to perform single-point evaluation for $\widetilde{O}(\sqrt{N})$ different PPPRF keys where N is the database size. With the current construction, each evaluation takes time $m > n = \sqrt{N}$ (the input domain size of the PPPRF we need for the PIR construction is $\sqrt{N}$). As a result, the client would be subject to $\Omega(N)$ computation per query.

2.2 Efficiency Upgrade: Forest-Based PPPRF

To overcome the efficiency issue, the intuitive idea is to use a tree structure similar to the Goldreich, Goldwasser, and Micali [GGM86] PRF construction. Say the inefficient PPPRF takes quasi-linear time (w.r.t. the input domain size) to evaluate a single point. Consider a tree of n leaves where each internal node has a wide fan-out of $\gamma = n^\epsilon$ for some small constant $\epsilon < 1$. Each internal node in the tree is associated with a PPPRF key. The root's key is sampled uniformly at random. The key of the i-th child of an internal node is computed by evaluating the PPPRF at i with the parent's key. Similarly, we use the last level's keys to derive the leaves' values. To evaluate the tree-based PPPRF at any point x, we only need to evaluate the keys along the path from the root to the x-th leaf. The evaluation time is $\widetilde{O}(\log_\gamma n \cdot \gamma) = \widetilde{O}(n^\epsilon/\epsilon)$, which is better than directly using the inefficient PPPRF on input domain $[n]$ where the evaluation time is $\Omega(n)$.

To program the tree-based PPPRF at x, we program the inefficient PPPRF keys along the path from the x-th leaf to the root. However, even if the inefficient PPPRF provides perfect programming privacy, this construction still fails to provide programming privacy. Since a normal key and a programmed key of the inefficient PPPRF are actually distinguishable by their key types, an adversary can identify a path from the root to a leaf where all the associated keys are programmed keys. This leaks the information about the programmed point.

We address this issue by 1) carefully designing the key structure associated with each node; 2) hiding the actual programmed tree among a forest of Z trees. We now provide a high-level description of our construction.

Tree structure. Let the input domain be $\{0, 1, \ldots, n-1\}$. Each tree has n leaves and a wide fan-out of $\gamma = n^\epsilon$ for some constant $\epsilon \in (0,1)$. Each node is associated with a key tuple $(\mathsf{msk}, \mathsf{sk}, \tau)$

where msk denotes an unprogrammed PPPRF key, sk denotes a programmed PPPRF key, and $\tau \in \{0,1\}$ is a bit that chooses which of the two keys will be used to evaluate the key tuples of the immediate children. Specifically, if $\tau = 0$, we use $\mathsf{PPPRF.Eval}(\mathsf{msk}, i)$ to compute the i-th child's key tuple where $i \in \{0, 1, \ldots, \gamma - 1\}$; and if $\tau = 1$, we use $\mathsf{PPPRF.PEval}(\mathsf{sk}, i)$ instead. Henceforth, the bit τ is also called the *type* of the nodes.

A tree with the root key $(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0)$ can only be programmed at the point $x \in \{0, 1, \ldots, n-1\}$ if every node the path from the root to the leaf x is of the type $\tau = 0$. Henceforth, let $D = \lceil \log_\gamma n \rceil = \Theta(1/\epsilon)$ be the depth of the tree. To program the key forcing its outcome at x to be v, let $(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0)$, $\ldots$, $(\mathsf{msk}_{D-1}, \mathsf{sk}_{D-1}, \tau_{D-1})$ be the original key tuples from the root to the leaf x. Henceforth, write $x = (x_0, \ldots, x_{D-1})$ in base-γ format. We start at the leaf level $D-1$ and work our way backwards. We first program msk_{D-1} forcing its outcome at x_{D-1} to be v, resulting in the programmed key sk'_{D-1}. We then program msk_{D-2} forcing its outcome at x_{D-2} to be $(\mathsf{msk}'_{D-1}, \mathsf{sk}'_{D-1}, 1)$ where msk'_{D-1} is chosen freshly at random. Suppose the resulting key is sk'_{D-2}. Next, we program msk_{D-3} forcing its outcome at x_{D-3} to be $(\mathsf{msk}'_{D-2}, \mathsf{sk}'_{D-2}, 1)$ where msk'_{D-2} is chosen freshly at random, and so on. At the end, we output the new key $(\mathsf{msk}'_0, \mathsf{sk}'_0, 1)$ at the root.

Forest-based PPPRF. Observe that with a single tree, if we sample a random root key, the probability that it can be programmed at a fixed point x is roughly $1/2^D = \Theta(1)$, since the type τ at each node is (pseudo-)random. This motivates our forest-based PPPRF construction: we will have Z independent trees, and the PPPRF's evaluation outcome at x is the XOR of all trees' evaluation outcomes at x. Programming at x will be successful as long as one of the trees can support programming at x. For programming privacy, consider that roughly $Z/2^D$ of the trees have type 1 on the entire path leading to the leaf x. So if the underlying PPPRF tree had negligible privacy loss, then we would get roughly $\sqrt{\frac{2^D}{Z}} = \Theta\left(\frac{1}{\sqrt{Z}}\right)$ privacy loss with a forest of Z trees. In our PIR application, we only need PPPRF with $1/\mathsf{poly}\log\lambda$ security, and thus we only need Z to be polylogarithmically large. The formal description of our forest-based construction is given in Section 6.2.

Efficiency. With a fan-out of n^ϵ, the tree has a constant depth of $D = \Theta(1/\epsilon)$. Keep in mind that for the underlying PPPRF, a programmed key is $O(\lambda \cdot \log\gamma)$ times longer than its output length ℓ_{out}. This means that in our tree construction, we suffer from a multiplicative blowup of $O(\lambda \cdot \log\gamma)$ with every level of the tree, and the key length at the root is $\ell_0 := (\lambda \cdot \log\gamma)^{O(1/\epsilon)} \cdot \ell_{\text{out}} = (\epsilon \cdot \lambda \cdot \log n)^{O(1/\epsilon)} \cdot \ell_{\text{out}}$. Because of this cumulative blowup, we want to choose $\epsilon \in (0,1)$ to be a constant, such that the key length at the root ℓ_0 is upper bounded by $\mathsf{poly}(\lambda, \log n)$.

Finally, with the tree-based construction, the input domain at each level of the tree is only $\gamma = n^\epsilon$ rather than n. With more careful accounting, we show in Section 6.2 that the evaluation time for one tree is upper bounded $n^\epsilon \cdot \mathsf{poly}(\lambda, \log n, \ell_{\text{out}})$. As mentioned earlier, we only need Z to be polylogarithmically large for the PIR application we care about, so the evaluation time of the entire forest-based PPPRF is $n^\epsilon \cdot \mathsf{poly}(\lambda, \log n, \ell_{\text{out}})$.

2.3 New Definitions and Proof Techniques: Functional Programming Privacy

Proving the security of our forest-based PPPRF turns out to be tricky. The straightforward idea — using a hybrid argument to replace the programmed keys with simulated keys one by one from the leaf to the root — runs into a circularity issue. Recall that the standard programming privacy requirement of PPPRF states that if we sample a random normal key, and then program it at an adversarially specified point x to a random value r, the programmed key is indistinguishable from a

simulated one [PS18]. If we were to do a hybrid argument and work our way backwards from leaf to root, we immediately encounter the following problem. Let $(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0)$, ..., $(\mathsf{msk}_{D-1}, \mathsf{sk}_{D-1}, \tau_{D-1})$ be the original key tuples from the root to the leaf $x = (x_0, \ldots, x_{D-1})$ expressed in base-γ format. Suppose we now program msk_{D-1} at the point x_{D-1} to a random outcome. We cannot directly use the security of the PPPRF because msk_{D-1} is not sampled completely at random. Further, even though it seems like msk_{D-1} is pseudorandom, we cannot just replace msk_{D-1} with random either, because we have not removed msk_0 from the experiment yet and msk_{D-1} is a function of msk_0.

We devise novel techniques to get around this circularity issue. Specifically, we strengthen the security definition of the underlying PPPRF to a new notion called *functional programming privacy*. In functional programming privacy, the adversary specifies a point x to be programmed at, and a function f, and the programmed value is the function f applied to the original outcome of the PPPRF at x. Functional programming privacy says that if f produces random outputs upon random inputs, then the adversary cannot distinguish a programmed key from a simulated key.

We can further extend the notion to support functions f whose outputs are approximately random upon receiving random inputs. Formally, given some efficiently computable, possibly randomized function $f : \{0,1\}^\ell \to \{0,1\}^\ell$, we say that f is *δ-random-to-random*, if upon receiving a random ℓ-bit input, the output of f is δ-computationally indistinguishable from random. Given this, we can define functional programming privacy as follows:

Definition 2.1 (Functional programming privacy of PPPRF)**.** We say that a PPPRF with ℓ_{in}-bit inputs and ℓ_{out}-bit outputs satisfies δ-general functional programming privacy, iff there exists a PPT simulator Sim, such that the following experiments are $(\delta + \delta')$-computationally indistinguishable to any non-uniform PPT adversary $\mathcal{A}$:

- $\mathsf{RealProgPriv}$: $\mathcal{A}$ specifies x and f where f is required to be PPT and δ'-random-to-random. Sample a random PPPRF key msk, program msk at x to be $f(\mathsf{PPPRF.Eval}(\mathsf{msk}, x))$, and return the programmed key to $\mathcal{A}$.
- $\mathsf{IdealProgPriv}$: $\mathcal{A}$ specifies x and f where f is required to be PPT and δ'-random-to-random. Return the simulated key generated by Sim to $\mathcal{A}$.

Upgrading the underlying PPPRF to functional programming privacy. First, we need to upgrade our underlying inefficient PPPRF to enjoy functional programming privacy. The details are described in Section 5. At a high level, we first prove that (a slight variant of) the construction in Section 2.1 satisfies $1/\mathsf{poly}$-functional programming privacy w.r.t. to the XOR family of functions (Section 5.1 and appendix B.1). Next, we describe a simple upgrade such that the resulting PPPRF satisfies $1/\mathsf{poly}$-functional programming privacy w.r.t. general functions that are approximately random-to-random (Section 5.2 and appendix B.2).

Proving the forest-based PPPRF using functional programming privacy. The security proof for the forest-based PPPRF consists of two main parts:

1. First, we prove a technical lemma regarding the security of a single tree (Theorem 6.1), and the proof crucially relies on the fact that the underlying PPPRF satisfies $1/\mathsf{poly}$-*functional* programming privacy (Appendix C.1). The proof relies on an inductive argument to show that given the root key $(\mathsf{msk}_0, \_, \_)$, and some point x_0 to program msk_0 at, the value we want to program to can be viewed as some (complex-to-state) function of $\mathsf{Eval}(\mathsf{msk}_0, x_0)$, and moreover, this function is approximately random-to-random.
2. Next, we prove that the Z-fold repetition provides sufficient decoy to approximately mask the point that is being programmed (Appendix C.2).

2.4 PIR Security Amplification

So far, we have constructed a weakly secure PPPRF for small domains. We first use the weakly secure PPPRF to get a weakly secure PIR (for both the 1-server and 2-server settings), and then we use a security amplification technique to boost the security failure probability to negligibly small.

2.4.1 Weakly Secure PIR from Weakly Secure PPPRF

Using ideas similar to prior works [ZPSZ24, GZS24], we can construct a suitable Privately Programmable Pseudorandom Set (PPPS) using PPPRF. We can then use the resulting PPPS to construct a 1-server or 2-server preprocessing PIR (Section 7.2). Specifically, we will instantiate the forest-based PPPRF with $1/\mathsf{poly}\log\lambda$-programming privacy. The resulting 1-server or 2-server proceprossing PIR schemes also have $1/\mathsf{poly}\log\lambda$-security. Moreover, each server stores only the original database, and all other costs match the bounds stated in Theorem 1.2 and Theorem 1.3.

2.4.2 Security Amplification

We want to achieve security amplification using a similar technique as Boyle et al. [BGIK22]. However, upon a closer inspection, we found that directly using Boyle et al. [BGIK22]'s techniques in our setting would result in an unacceptable blow up in the costs (see Theorem E.1). To fix this, we strengthen the underlying weakly-secure PIR scheme with additional decoy queries to satisfy a slightly stronger security notion, and then use Locally-Decodable Codes (LDC) to amplify the security.

The idea is that by adding suitable decoy queries for each real query of the weakly-secure PIR scheme, we can achieve the following leaky PIR scheme that provides a slighly stronger security guarantee called strongly δ-security: for each query, with probability $1-\delta$, the scheme does not disclose any information (in a computational sense); and with probability δ, some information about the query is leaked — the precise definition is given in Appendix E.2, and the proof relies on the hardcore lemma for computational indistinguishability [MT10].

We then use a Locally-Decodable Code (LDC) to perform security amplification, similar to [BGIK22]. We can encode the original DB of size N into a codeword $\widetilde{\mathsf{DB}}$ of size $N^{1+\epsilon}$ where ϵ is an arbitrarily small constant. Now, to learn any $\mathsf{DB}[i]$, we can alternatively access polylogarithmically many locations denoted $w_1, \ldots, w_Q$ in the codeword $\widetilde{\mathsf{DB}}$, and the client can recover $\mathsf{DB}[i]$ from $\widetilde{\mathsf{DB}}[w_1], \ldots, \widetilde{\mathsf{DB}}[w_Q]$ with the LDC decoding algorithm. Intuitively, if we use the leaky PIR scheme for the queries, we are leaking roughly $\delta \cdot Q$ queries. The LDC code ensures that as long as we carefully pick the parameters, even leaking those $\delta \cdot Q$ queries to the server will not reveal any information about the true query index i. We finally show that we can set $\delta = 1/\mathsf{poly}_1\log\lambda$, $Q = \mathsf{poly}_2\log\lambda$ and a Reed-Muller code with smoothness $\sigma = \mathsf{poly}_3\log\lambda$ for suitable choices of $\mathsf{poly}_1, \mathsf{poly}_2$, and poly_3 to get a computationally secure PIR scheme with polylogarithmic overhead in the costs.

3 Preliminaries

3.1 δ-Computational Indistinguishability

Definition 3.1 (δ-computational indistinguishability)**.** Let $\delta(\lambda) : \mathbb{N} \to [0, 1)$ be a function of $\lambda \in \mathbb{N}$. We say that two probability ensembles $D_1(1^\lambda)$ and $D_2(1^\lambda)$ indexed by the security parameter λ are δ-computationally indistinguishable, denoted by $D_1 \overset{\delta}{\approx}_c D_2$, if for any non-uniform probabilistic

polynomial-time (PPT) distinguisher $\mathcal{A}$, there exists some negligible function $\mathsf{negl}(\lambda)$ such that for sufficiently large $\lambda \in \mathbb{N}$,

$$\left|\Pr[\mathcal{A}(1^\lambda, D_1(1^\lambda)) = 1] - \Pr[\mathcal{A}(1^\lambda, D_2(1^\lambda)) = 1]\right| \leq \delta(\lambda) + \mathsf{negl}(\lambda).$$

For the special case when $\delta(\lambda) = 0$, we also say that D_1 and D_2 are computationally indistinguishable, denoted $D_1 \approx_c D_2$.

The following simple facts hold for δ-computational indistinguishability.

- *Transitivity.* Let m be polynomially bounded in λ. Given m probability ensembles $D_1, D_2, \ldots, D_m$ such that for every $i \in [m-1]$, $D_i \overset{\delta}{\approx}_c D_{i+1}$, then D_1 and D_m are $m\delta$-computationally indistinguishable.
- *Post-processing.* If $D_1 \overset{\delta}{\approx}_c D_2$ and f is a PPT function, then $f(D_1) \overset{\delta}{\approx}_c f(D_2)$.

3.2 Building Block: Puncturable PRF

A puncturable PRF [BW13, BGI14] is a pseudorandom function (PRF) that can supports puncturing of a normal key at a specified point, resulting in a punctured key. The punctured key allows one to evaluate the PRF at all points except the punctured point.

A puncturable PRF consists of the following possibly randomized algorithms:

- $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$: The key generation algorithm takes a security parameter λ, the input length ℓ_{in}, the output length ℓ_{out}, and samples a normal PRF key denoted sk.
- $y \leftarrow \mathsf{Eval}(\mathsf{sk}, x)$: The evaluation algorithm takes a normal key sk and an input $x \in \{0,1\}^{\ell_{\text{in}}}$ and outputs the evaluation outcome $y \in \{0,1\}^{\ell_{\text{out}}}$.
- $\mathsf{sk}' \leftarrow \mathsf{Punct}(\mathsf{sk}, x)$: The Punct algorithm takes a normal key sk, a point $x \in \{0,1\}^{\ell_{\text{in}}}$, and outputs a punctured key sk'.
- $\mathsf{PEval}(\mathsf{sk}', x)$: takes in a punctured key sk' and an input $x \in \{0,1\}^{\ell_{\text{in}}}$, and outputs either an evaluation outcome $y \in \{0,1\}^{\ell_{\text{out}}}$ or $\perp$ indicating failure.

Correctness. We say that a puncturable PRF is correct, iff for any λ, any ℓ_{in} and ℓ_{out}, any $x \in \{0,1\}^{\ell_{\text{in}}}$, the following holds:

$$\Pr\left[\begin{array}{c} \mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}}), \\ \mathsf{sk}' \leftarrow \mathsf{Punct}(\mathsf{sk}, x) \end{array} : \begin{array}{c} \mathsf{PEval}(\mathsf{sk}', x) = \perp \quad \text{and} \\ \forall x' \neq x : \mathsf{PEval}(\mathsf{sk}', x') = \mathsf{Eval}(\mathsf{sk}, x') \end{array}\right] = 1$$

We require that PEval at the punctured point always gives $\perp$; and besides the punctured point, all other points' evaluations are unaffected by the puncturing.

Security. We say that a puncturable PRF scheme is secure, iff it satisfies the standard pseudorandomness property of an ordinary PRF, and moreover, for every $\ell_{\text{in}}(\lambda)$ and $\ell_{\text{out}}(\lambda)$ that are polynomially bounded in λ, the following two experiments $\mathsf{Real}^{\mathcal{A}}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ and $\mathsf{Ideal}^{\mathcal{A}}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ are computationally indistinguishable for any non-uniform PPT adversary $\mathcal{A}$:

- $\mathsf{Real}^{\mathcal{A}}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$: $\mathcal{A}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ outputs $x \in \{0,1\}^{\ell_{\text{in}}}$; let $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda)$, $\mathsf{sk}' \leftarrow \mathsf{Punct}(\mathsf{sk}, x)$, and let $y \leftarrow \mathsf{Eval}(\mathsf{sk}, x)$; output $\mathcal{A}(\mathsf{sk}', y)$.

- $\mathsf{Ideal}^{\mathcal{A}}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$: $\mathcal{A}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ outputs $x \in \{0,1\}^{\ell_{\text{in}}}$; let $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda)$, $\mathsf{sk}' \leftarrow \mathsf{Punct}(\mathsf{sk}, x)$, and let $y \xleftarrow{\$} \{0,1\}^{\ell_{\text{out}}}$; output $\mathcal{A}(\mathsf{sk}', y)$.

Intuitively, given a key punctured at some point x, a computationally bounded adversary cannot distinguish the original PRF's evaluation at x from random.

Earlier works [BW13, BGI14] showed that the tree-based PRF construction from pseudorandom generators (PRGs) [GGM86] implies a puncturable PRF construction. Specifically, a normal key has length $O_\lambda(1)$, the length of a punctured key and the runtime of the puncturing procedure is bounded by $O_\lambda(\ell_{\text{in}})$, and both normal evaluation Eval and punctured evaluation PEval takes time at most $O_\lambda(\ell_{\text{in}} + \ell_{\text{out}})$, where $O_\lambda(\cdot)$ hides $\mathsf{poly}(\lambda)$ terms.

4 Definitions

4.1 Privately Programmable Pseudorandom Functions

Intuitively, a Privately Programmable Pseudorandom Function (PPPRF) is a PRF with an additional programming functionality that allows one to force the PRF to output a specific value at a specific point.

Syntax. A programmable PRF is a tuple $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Prog}, \mathsf{PEval})$ of possibly randomized algorithms with the following syntax:

- $\mathsf{Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$: given the security parameter λ, the input and output lengths ℓ_{in} and ℓ_{out}, output a master secret key msk.
- $\mathsf{Eval}(\mathsf{msk}, x)$: given the master secret key msk and an input $x \in \{0,1\}^{\ell_{\text{in}}}$, output the evaluation outcome $v \in \{0,1\}^{\ell_{\text{out}}}$.
- $\mathsf{Prog}(\mathsf{msk}, x, v)$: given the master secret key msk and the programming point x and the programmed value v, output a programmed key sk.
- $\mathsf{PEval}(\mathsf{sk}, x)$: given a programmed key sk and an input $x \in \{0,1\}^{\ell_{\text{in}}}$, output the evaluation outcome $v \in \{0,1\}^{\ell_{\text{out}}}$.

Programming correctness. The correctness definition requires that the programming procedure Prog correctly changes the evaluation at the programmed point, without affecting anything else. Formally, a programmable PRF satisfies correctness if for all $\ell_{\text{in}}(\cdot)$ and $\ell_{\text{out}}(\cdot)$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all λ, for all $x \in \{0,1\}^{\ell_{\text{in}}}$ and $v \in \{0,1\}^{\ell_{\text{out}}}$, we have the following:

$$\Pr\left[\begin{array}{c} \mathsf{msk} \leftarrow \mathsf{Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}}), \\ \mathsf{sk} \leftarrow \mathsf{Prog}(\mathsf{msk}, x, v), \end{array} : \begin{array}{c} \mathsf{PEval}(\mathsf{sk}, x) = v \ \text{ and} \\ \forall x' \neq x, \mathsf{PEval}(\mathsf{sk}, x') = \mathsf{Eval}(\mathsf{msk}, x') \end{array}\right] \geq 1 - \mathsf{negl}(\lambda).$$

Functional programming privacy. The standard security notion for privately programmable PRFs [PS18] requires that the programmed key hides the programmed point x — note that this cannot be achieved if we program x to some known value v, since the adversary can always test the programmed key to see where it evaluates to v. Therefore, the standard "private programmability" notion [PS18] requires that the programmed key hides the programmed point x, as long as we program x to a randomly chosen value.

For technical reasons, we need a stronger notion of private programmability called functional programming privacy. Specifically, we require that the programmed key hides the programmed

$\underline{\mathsf{RealProgPriv}^{\mathcal{A}}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})}$:	$\underline{\mathsf{IdealProgPriv}^{\mathcal{A},\mathsf{Sim}}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})}$:
$x, f \leftarrow \mathcal{A}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$	$x, f \leftarrow \mathcal{A}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$
$\mathsf{msk} \leftarrow \mathsf{PPPRF.Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$	$\mathsf{sk} \leftarrow \mathsf{Sim}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$
$v \leftarrow f(\mathsf{Eval}(\mathsf{msk}, x))$	output $\mathcal{A}(\mathsf{sk})$
$\mathsf{sk} \leftarrow \mathsf{Prog}(\mathsf{msk}, x, v)$	
output $\mathcal{A}(\mathsf{sk})$	

Figure 1: Security experiments for functional programming privacy, where the adversary $\mathcal{A}$ is stateful.

point x when we program x to some function f of the original PRF's outcome at x, as long as the function f's output distribution is indistinguishable from random upon receiving a uniform random input. Our formal definition below (Theorem 4.2) is more general: we not only permit the function f to be δ away from random upon receiving a random input, but also allow for a privacy error in the functional programming privacy notion.

Definition 4.1 (δ-random-to-random functions)**.** Given a function $f : \{0,1\}^k \to \{0,1\}^k$, we say that f is δ-random-to-random, iff the following holds where $\mathcal{U}_k$ denotes the uniform distribution over $\{0,1\}^k$:

$$f(\mathcal{U}_k) \overset{\delta}{\approx}_c \mathcal{U}_k$$

For the special case when $\delta = 0$, we also say that f is random-to-random.

A function family $\mathcal{F}$ parametrized by λ and $\ell_{\text{out}}(\cdot)$ that map $\{0,1\}^{\ell_{\text{out}}(\lambda)}$ to $\{0,1\}^{\ell_{\text{out}}(\lambda)}$ is said to be δ-random-to-random, if every $f \in \mathcal{F}$ is δ-random-to-random where δ is a function of λ and ℓ_{out} whose output range is $[0, 1)$.

Definition 4.2 (Functional programming privacy)**.** Let $\mathcal{F}$ be a δ'-random-to-random family of PPT functions parametrized by λ and $\ell_{\text{out}}(\cdot)$, which map $\{0,1\}^{\ell_{\text{out}}(\lambda)}$ to $\{0,1\}^{\ell_{\text{out}}(\lambda)}$. Let δ be a function in λ, ℓ_{in}, and ℓ_{out} whose output range is $[0, 1)$. We say that a programmable PRF $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Prog}, \mathsf{PEval})$ satisfies δ-functional programming privacy w.r.t. $\mathcal{F}$, if there exists a PPT simulator Sim, such that for any non-uniform PPT admissible adversary $\mathcal{A}$, for any ℓ_{in} and ℓ_{out} that are polynomially bounded functions in λ,

$$\mathsf{RealProgPriv}^{\mathcal{A}}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}}) \overset{\delta+\delta'}{\approx}_c \mathsf{IdealProgPriv}^{\mathcal{A}}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$$

where an adversary is admissible if it outputs a function $f \in \mathcal{F}$ with probability 1.

For the special case when $\delta = 0$, we simply say that the programmable PRF satisfies functional programming privacy w.r.t. $\mathcal{F}$.

We will specifically care about the following special cases:

- **Standard private programmability.** The standard private programmability notion [PS18] is a special case of Theorem 4.2 when the family $\mathcal{F}$ consists of only a single function $f_{\text{rand}}(\cdot)$ that simply ignores the input and samples a uniform random string from $\{0,1\}^{\ell_{\text{out}}}$.
- **XOR functional programming privacy.** We say that a PRF satisfies δ-XOR functional programming privacy, if it satisfies δ-functional programming privacy w.r.t. the following XOR function family:

$$\mathcal{F}_{\text{xor}} = \{f^\Delta(\cdot)\}_{\Delta \in \{0,1\}^{\ell_{\text{out}}}}, \text{ where } f^\Delta(v) = v \oplus \Delta.$$

- **General functional programming privacy.** We say that a PRF satisfies δ-general functional programming privacy, iff for any $\delta'(\cdot)$, for any δ'-random-to-random family $\mathcal{F}$ of PPT functions, the scheme satisfies δ-functional programming privacy w.r.t. $\mathcal{F}$.

5 Inefficient, Small-Domain Privately Programmable PRF

5.1 PRF with XOR Functional Programming Privacy

Boyle et al. [BGIK22] propose an inefficient 1/poly-secure programmable distributed point function for small domains. It is not hard to modify their construction and obtain an inefficient 1/poly-secure privately programmable PRF (PPPRF) for small domains, satisfying XOR-functional programming privacy.

PRF with XOR functional programming privacy

Notation: let pPRF be a puncturable PRF, and let xPRF be a regular PRF. The input domain is $\{0, 1, \ldots, n-2\}$ where $n = 2^{\ell_{\text{in}}}$. Let $m = \mathsf{poly}(n)$ be a power of 2.

$\underline{\mathsf{Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}} = 1)}$:

- let $\mathsf{sk} \leftarrow \mathsf{pPRF.Gen}(1^\lambda, \log_2 m, \ell_{\text{in}})$; $\mathsf{xsk} \leftarrow \mathsf{xPRF.Gen}(1^\lambda, \ell_{\text{in}}, 1)$; $s \xleftarrow{\$} \{0, 1, \ldots, n-1\}$;
- output $\mathsf{msk} = (\mathsf{psk}, \mathsf{xsk}, s)$.

$\underline{\mathsf{Eval}(\mathsf{msk} = (\mathsf{psk}, \mathsf{xsk}, s), x)}$:

- for $i \in \{0, 1, \ldots, m-1\}$, let $w_i := (\mathsf{pPRF.Eval}(\mathsf{psk}, i) + s) \mod n$;
- let $\mathsf{Parity}(x) = \oplus_{i \in \{0,1,\ldots,m-1\}} \mathbb{1}(w_i = x)$ where $\mathbb{1}(\cdot)$ is the indicator function;
- output $\mathsf{Parity}(x) \oplus \mathsf{xPRF.Eval}(\mathsf{xsk}, x)$,

$\underline{\mathsf{Prog}(\mathsf{msk} = (\mathsf{psk}, \mathsf{xsk}, s), x, b)}$:

- let $v \leftarrow \mathsf{Eval}(\mathsf{msk}, x)$;
- let $x' = x$ if $v \neq b$, else let $x' = n - 1$; *// $n-1$ is the special bin*
- uniformly sample r from $\{0, 1, \ldots, m-1\}$ subject to $\mathsf{pPRF.Eval}(\mathsf{psk}, r) + s = x' \mod n$; output $\perp$ if no such r exists;
- output $\mathsf{sk} = (\mathsf{pPRF.Punct}(\mathsf{psk}, r), \mathsf{xsk}, s)$.

$\underline{\mathsf{PEval}(\mathsf{sk} = (\mathsf{psk}', \mathsf{xsk}, s), x)}$:

- for $i \in \{0, 1, \ldots, m-1\}$, let $w_i := (\mathsf{pPRF.PEval}(\mathsf{psk}', i) + s) \mod n$ — if $\mathsf{pPRF.PEval}(\mathsf{psk}', i) = \perp$, we define $w_i = \perp$;
- output $\mathsf{Parity}(x) \oplus \mathsf{xPRF.Eval}(\mathsf{xsk}, x)$ where $\mathsf{Parity}(x)$ is defined earlier in $\mathsf{Eval}(\mathsf{msk}, x)$.

Figure 2: Small-domain, 1-bit output PRF with XOR functional programming privacy.

Intuition. Suppose we want a PPPRF with input length ℓ_{in} and output length $\ell_{\text{out}} = 1$. We will assume that the input domain size $n = 2^{\ell_{\text{in}}}$ is polynomially bounded in λ. As building blocks, we

rely on a puncturable PRF denoted pPRF, and another regular PRF (used for padding) henceforth denoted xPRF.

The PPPRF's master secret key $\mathsf{msk} := (\mathsf{psk}, \mathsf{xsk}, s)$ consists of psk which is the secret key of the underlying pPRF, a padding secret key denoted xsk, and a random offset $s \in \{0, 1, \ldots, n-1\}$. Henceforth, we will *treat the point $n-1$ as a special point reserved for programming, and remove it from the input domain.* In other words, we assume that the input domain to our PPPRF is $\{0, 1, \ldots, n-2\}$ where $n = 2^{\ell_{\text{in}}}$. To evaluate the PPPRF at some point $x \in [n-1]$, we will use the underlying pPRF to throw $m = \mathsf{poly}(n)$ balls into n bins numbered $0, 1, \ldots, n-1$. Specifically, for $i \in \{0, 1, \ldots, m-1\}$, $(\mathsf{pPRF.Eval}(\mathsf{psk}, i) + s) \mod n$ decides which bin ball i lands in. The evaluation outcome $\mathsf{PPPRF.Eval}(\mathsf{msk}, x)$ at point $x \in [n-1]$ is defined to be $\mathsf{Parity}(x) \oplus \mathsf{xPRF}(\mathsf{xsk}, x)$ where $\mathsf{Parity}(x)$ denotes the number of balls in bin x mod 2. The padding $\mathsf{xPRF}(\mathsf{xsk}, x)$ is used to make the outcome pseudorandom. To program the master secret key $\mathsf{msk} = (\mathsf{sk}, s)$ at some point $x \in [n-1]$ to the bit $b \in \{0, 1\}$, we check if b is equal to the original evaluation outcome. If so, let r be a random ball that lands in the special bin $n-1$; otherwise, r be a random ball that lands in bin x. Now, we compute $\mathsf{psk}' \leftarrow \mathsf{PRF.Punct}(\mathsf{psk}, r)$, and output the punctured key $(\mathsf{psk}', \mathsf{xsk}, s)$.

Such a PPPRF construction suffers from two caveats:

- *1/poly-security.* Programming a key at the point x slightly skews the distribution of the load of bin x or the special bin $n-1$, which gives the adversary a small advantage in the functional programming privacy security game. In Appendix B.1, we prove that the adversary has $1/\mathsf{poly}(n)$ advantage in distinguishing a programmed key and a simulated key.
- *Small domain restriction and lack of efficiency.* The scheme only supports polynomially sized input domain and is relatively inefficient because the evaluation procedure Eval and the programming procedure Prog take $O_\lambda(m)$ time. As we will show later, to achieve δ-security, we need to set $m = n/\delta^2$.

Formal description. We give a formal description of the scheme in Figure 2. When instantiated Goldreich, Goldwasser, and Micali [GGM86] as the underlying puncturable PRF, the resulting scheme has the following performance:

- A normal key has size λ;
- A programmed key has size $O(\lambda \log n)$;
- Eval, PEval, and Prog take $O(m \cdot \log n) \cdot T_{\text{PRG}}(\lambda)$ time where $T_{\text{PRG}}(\lambda) = \mathsf{poly}(\lambda)$ is the time it takes to evaluate a PRG that maps λ bits to 2λ bits. Theorem 5.1 below shows that to get δ-security, we need to set $m = n/\delta^2$.

Security. In Appendix B.1, we prove the following theorem.

Theorem 5.1 (Small-domain PPPRF with XOR functional programming privacy)**.** *Suppose that $m = \omega(n \cdot \log \lambda)$ and m is upper bounded by $\mathsf{poly}(n)$. The PPPRF construction in Figure 2 satisfies correctness, and $\sqrt{n/m}$-functional programming privacy w.r.t. the family $\mathcal{F}_{\text{xor}}$.*

For the special case when the underlying pPRF is instantiated with Goldreich, Goldwasser, and Micali [GGM86], the simulator Sim in $\mathsf{IdealProgPriv}$ can simply output a random string of appropriate length, as stated in the following corollary (whose proof is in Appendix B.1).

Corollary 5.2. *Suppose $m = \omega(n \cdot \log \lambda)$ and is upper bounded by $\mathsf{poly}(n)$, and the pPRF is instantiated with Goldreich, Goldwasser, and Micali [GGM86]. Then, our PPPRF satisfies $\sqrt{n/m}$ functional programming privacy w.r.t. $\mathcal{F}_{\text{xor}}$ where the simulator Sim in $\mathsf{IdealProgPriv}$ outputs a random string of appropriate length.*

5.2 Upgrade to General Functional Programming Privacy

To upgrade to general functional programming privacy, we consider ℓ_{out} parallel instances of the underlying PPPRF which has 1-bit outcome and supports XOR functional programming privacy. Further, we XOR the outcome of the PRF with a random pad (denoted pad) of ℓ_{out} bits. Due to space constraint, we describe the algorithm in Figure 7 of Appendix B.

Theorem 5.3 (PPPRF with general functional programming privacy)**.** *Suppose the underlying* PPPRF *satisfies programming correctness and δ-XOR functional programming privacy. Then, the construction in Figure 7satisfies programming correctness and $(\ell_{\text{out}} \cdot \delta)$-general functional programming privacy. Further, the scheme achieves the following performance bounds where $n = 2^{\ell_{\text{in}}}$ denotes the input domain size:*

- *A normal key has size $O(\ell_{\text{out}} \cdot \lambda)$;*
- *A programmed key has size $O(\lambda \cdot \ell_{\text{out}} \cdot \log n)$;*
- Eval, PEval, *and* Prog *takes $O(\ell_{\text{out}} \cdot n \cdot \log n/\delta^2) \cdot T_{\text{PRG}}(\lambda)$ time.*

The random pad is the critical component of the upgrade. It ensures that the output at the programming point is perfectly random, regardless of the sampled PPPRF keys. Therefore the PPPRF keys can be sampled independently from the real output v, and v alone decides $v \oplus f(v)$ which will be xor-ed at the programming point. This allows us to invoke the XOR functional programming privacy property.

Using Theorem 5.2 as the underlying PPPRF, and setting $\widetilde{\delta} = \ell_{\text{out}} \cdot \delta$, we get:

Corollary 5.4 (PRF with general functional programming privacy)**.** *Assume the existence of one-way functions. There exists a PRF that satisfies $\widetilde{\delta}$-general functional programming privacy w.r.t. a simulator* Sim *that outputs a random string of appropriate length. Further, the scheme satisfies the following performance bounds where $n = 2^{\ell_{\text{in}}}$:*

- *A normal key has size $O(\ell_{\text{out}} \cdot \lambda)$;*
- *A programmed key has size $O(\lambda \cdot \ell_{\text{out}} \cdot \log n)$;*
- Eval, PEval, *and* Prog *take $O(n \cdot \log n \cdot \ell_{\text{out}}^3/\widetilde{\delta}^2) \cdot T_{\text{PRG}}(\lambda)$ time.*

6 Efficient Forest-Based PPPRF

One drawback of the construction in Section 5 is that programming and evaluation take polynomial time in the input domain size. In this section, we build a new PPPRF from OWF that has polylogarithmic programming and evaluation time. Our construction involves creating Z PPPRF trees, where each tree has a GGM-like structure, but the underlying pseudorandom generator (PRG) is now replaced with the inefficient PPPRF of Section 5.2. To evaluate the resulting PPPRF at x, we simply evaluate each PPPRF tree at x, and output the XOR of all Z results. Given a point x, each tree is programmable at x with constant probability. Therefore, to program the PPPRF at x, we will select a random tree that is programmable at x to program.

6.1 Subroutine: PPPRF Tree

Subroutine: $\mathsf{PPPRFTree}$

Notation. Let PPPRF^i be a PRF with δ-general functional programming privacy, input length $\lceil \log_2(\gamma+1) \rceil$, and output length ℓ_{i+1}. Let γ be the fan-out, and $D = \lceil \log_\gamma(2^{\ell_{\text{in}}}) \rceil$ be the depth of the tree.

$\underline{\mathsf{Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})}$: let $\mathsf{msk}_0 \xleftarrow{\$} \{0,1\}^\lambda$, $\mathsf{sk}_0 \xleftarrow{\$} \{0,1\}^{|\mathsf{sk}_0|}$, $\tau_0 \xleftarrow{\$} \{0,1\}$. Output $(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0)$.

$\underline{\mathsf{Eval}(\mathsf{msk} = (\mathsf{msk}_0, \mathsf{sk}_0, \tau_0), x)}$:

- Let $x = x_0 + x_1\gamma + \cdots + x_{D-1}\gamma^{D-1}$ be the base-γ represenation of x.
- For i from 0 to $D-2$:
  - if $\tau = 0$, let $(\mathsf{msk}_{i+1}, \mathsf{sk}_{i+1}, \tau_{i+1}) \leftarrow \mathsf{PPPRF}^i.\mathsf{Eval}(\mathsf{msk}_i, x_i)$.
  - else, let $(\mathsf{msk}_{i+1}, \mathsf{sk}_{i+1}, \tau_{i+1}) \leftarrow \mathsf{PPPRF}^i.\mathsf{PEval}(\mathsf{sk}_i, x_i)$.
- If $\tau_{D-1} = 0$, output $\mathsf{PPPRF}^i.\mathsf{Eval}(\mathsf{msk}_{D-1}, x_{D-1})$; else, output $\mathsf{PPPRF}^i.\mathsf{PEval}(\mathsf{sk}_{D-1}, x_{D-1})$.

$\underline{\mathsf{Prog}(\mathsf{msk} = (\mathsf{msk}_0, \mathsf{sk}_0, \tau_0), x, v)}$:

- Let $x = x_0 + x_1\gamma + \cdots + x_{D-1}\gamma^{D-1}$ be the base-γ representation of x.
- Check that under msk, the entire path to x is programmable, and abort outputting $\perp$ if the check fails. Specifically, let $(\mathsf{msk}_1, \mathsf{sk}_1, \tau_1), \cdots, (\mathsf{msk}_{D-1}, \mathsf{sk}_{D-1}, \tau_{D-1})$ be obtained as in $\mathsf{Eval}(\mathsf{msk}, x)$. Verify that $\tau_0 = \tau_1 = \tau_2 = \ldots = \tau_{D-1} = 0$.
- Let $v_D = v$.
- For i from $D-1$ downto 0:
  - Let $\mathsf{sk}'_i \leftarrow \mathsf{PPPRF}^i.\mathsf{Prog}(\mathsf{msk}_i, x_i, v_{i+1})$, $\mathsf{msk}'_i \xleftarrow{\$} \{0,1\}^\lambda$
  - Let $v_i = (\mathsf{msk}'_i, \mathsf{sk}'_i, 1)$.
- Output v_0.

Figure 3: $\mathsf{PPPRFTree}$ subroutine.

Tree structure. The idea is to use a GGM-like, wide-fan-out tree, but using the PPPRF of Section 5.2 as the underlying pseudorandomness generator. Henceforth let γ be the fan-out of the tree. We will choose $\gamma = n^{1/\epsilon}$ for some constant $\epsilon \in (0,1)$, so the tree has constant depth $O(1/\epsilon)$.

Each node u in the tree is associated with a key tuple denoted $(\mathsf{msk}_u, \mathsf{sk}_u, \tau_u)$, where msk_u denotes an unprogrammed PRF key, sk_u denotes a programmed PRF key, and $\tau_u \in \{0,1\}$ is a bit indicating whether the unprogrammed key msk_u or the programmed key sk_u is active. The active key should be used to generate the key tuples for the children of u. More formally, suppose u is not at the leaf level and let $u||i$ be the i-th child of u, then,

- if $\tau_u = 0$, then $(\mathsf{msk}_{u||i}, \mathsf{sk}_{u||i}, \tau_{u||i}) := \mathsf{PPPRF.Eval}(\mathsf{msk}_u, i)$;
- else $(\mathsf{msk}_{u||i}, \mathsf{sk}_{u||i}, \tau_{u||i}) := \mathsf{PPPRF.PEval}(\mathsf{sk}_u, i)$.

Notice that the nodes at the leaf level are no longer associated with keys, but simply just pseudorandom bit strings of length ℓ_{out}.

Length of keys at different levels. The keys have different lengths at the different levels. Henceforth suppose the leaf is at level $D-1$, and the root is at level 0. At the leaf level, the key tuple is upper bounded by $\ell_{D-1} := C\lambda \cdot \ell_{\text{out}} \cdot \log \gamma$ for some constant $C > 1$. At level $D-2$, we use a PPPRF that can generate a length-ℓ_{D-1} outcome, and thus a level-$(D-2)$ key tuple has length at most $\ell_{D-2} := (C\lambda \log \gamma)^2 \cdot \ell_{\text{out}}$. More generally, a level-$(D-i)$ key tuple has length at most $\ell_{D-i} := (C\lambda \log \gamma)^i \cdot \ell_{\text{out}}$. For convenience, henceforth we define $\ell_D := \ell_{\text{out}}$.

Construction. We describe our construction in Figure 3. We already explained the tree structure earlier and how to expand the keys associated with all nodes in the tree. We now explain the intuition behind programming. Let $x_0, \ldots, x_{D-1}$ be the base-γ representation of some point x. To program the tree-based PPPRF at some point x to the outcome v, let $(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0), \ldots, (\mathsf{msk}_{D-1}, \mathsf{sk}_{D_1}, \tau_{D-1})$ be the key tuples associated with the nodes on the path from the root to leaf x. We first check that the entire path to x is programmable, that is, $\tau_0 = \tau_1 = \ldots = \tau_{D-1} = 0$ — this happens with $1/2^D$ probability (ignoring negligible terms) which is constant assuming ϵ is constant (recall $D = \Theta(1/\epsilon)$). If the check fails, simply abort outputting $\perp$ indicating failure. We now program the leaf's normal key msk_{D-1} at x_{D-1} to v, resulting in the programmed key sk'_{D-1}. We now assign the leaf x a new key tuple $(\mathsf{msk}'_{D-1}, \mathsf{sk}'_{D-1}, 1)$ where msk'_{D-1} is freshly sampled. We next program the leaf's parent's normal key msk_{D-2} at point x to the value $(\mathsf{msk}'_{D-1}, \mathsf{sk}'_{D-1}, 1)$, resulting in the programmed key sk'_{D-2}; and we assign to the leaf's parent a new key tuple $(\mathsf{msk}'_{D-2}, \mathsf{sk}'_{D-2}, 1)$ where msk'_{D-2} is freshly sampled. This goes on until we arrive at the root, and we finally output the new key tuple at the root.

Henceforth, we use the following notation:

- $T^x(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0)$: let $(\mathsf{msk}_1, \mathsf{sk}_1, \tau_1), \cdots, (\mathsf{msk}_{D-1}, \mathsf{sk}_{D-1}, \tau_{D-1})$ be obtained as in $\mathsf{Eval}((\mathsf{msk}_0, \mathsf{sk}_0, \tau_0), x)$; output $(\tau_0, \ldots, \tau_{D-1})$.

That is, $T^x(\mathsf{msk})$ outputs whether each node is programmable on the path from the root to the leaf x. $T^x(\mathsf{msk}) = 0^D$ means that the entire path to x is programmable, and $T^x(\mathsf{msk}) = 1^D$ means that the entire path to x is unprogrammable.

Performance. We now analyze the performance of our $\mathsf{PPPRFTree}$.

- *Length of normal and programmed keys.* A normal key and a programmed key have the same length, which is bounded by $\ell_0 = (C \cdot \lambda \cdot \log \gamma)^D \cdot \ell_{\text{out}}$. For $\gamma = n^\epsilon$ where $\epsilon \in (0,1)$ is some constant, the key length is bounded by $(C \cdot \lambda \cdot \epsilon \cdot \log n)^{1+1/\epsilon} \cdot \ell_{\text{out}} = (\epsilon \cdot \lambda \cdot \log n)^{O(1/\epsilon)} \cdot \ell_{\text{out}}$.
- *Time of* Prog *and* Eval. The programming and evaluation time is dominated by the time consumed at the root level, that is, $O(\ell_0^3 \cdot \gamma \cdot \log \gamma / \delta^2) \cdot T_{\text{PRG}}(\lambda)$. For $\gamma = n^\epsilon$ for some constant $\epsilon \in (0,1)$, the above is bounded by $T_{\text{PRG}}(\lambda) \cdot (\epsilon \cdot \lambda \cdot \log n)^{O(1/\epsilon)} \cdot n^\epsilon \cdot \ell_{\text{out}}^3 / \delta^2$.

Security. We shall prove a technical lemma (Theorem 6.1) regarding the security of the $\mathsf{PPPRFTree}$ subroutine. This technical lemma will be later used to prove our forest-based PPPRF secure. We defer the proof to Appendix C.

Lemma 6.1 (Security of $\mathsf{PPPRFTree}$)**.** *Suppose that 2^D is polynomially bounded in λ, and for all $i \in \{0, \ldots, D-1\}$, PPPRF^i satisfies δ-general programming privacy and is instantiated with the scheme in Theorem 5.4. Then, for any non-uniform PPT adversary $\mathcal{A}$, for any $\ell_{\text{in}}(\cdot)$ and $\ell_{\text{out}}(\cdot)$ that are polynomially bounded in λ, the following two experiments are $2^D \cdot \log^{1.1} \lambda \cdot (\delta' + D\delta)$-computationally indistinguishable:*

- $\mathsf{Real}^{\mathcal{A}}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$: *$\mathcal{A}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ outputs $x \in \{0,1\}^{\ell_{\text{in}}}$ and a PPT, δ'-random-to-random function $g : \{0,1\}^{\ell_{\text{out}}} \to \{0,1\}^{\ell_{\text{out}}}$. Sample*

  $\mathsf{msk} \leftarrow \mathsf{PPPRFTree.Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ *subject to the predicate $T^x(\mathsf{msk}) = 0^D$, sample $v \xleftarrow{\$} \{0,1\}^{\ell_{\text{out}}}$, and let $\mathsf{sk} \leftarrow \mathsf{Prog}(\mathsf{msk}, x, g(v))$ where $v = \mathsf{Eval}(\mathsf{msk}, x)$. Output $\mathcal{A}(\mathsf{sk})$.*

- $\mathsf{Ideal}^{\mathcal{A}}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$: *$\mathcal{A}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ outputs $x \in \{0,1\}^{\ell_{\text{in}}}$ and a PPT function $g : \{0,1\}^{\ell_{\text{out}}} \to \{0,1\}^{\ell_{\text{out}}}$ that is δ'-random-to-random. Let sk be a random string of appropriate length such that $T^x(\mathsf{sk}) = 1^D$ holds. Output $\mathcal{A}(\mathsf{sk})$.*

In particular, if $\delta' = 0$, then the two experiments are $D \cdot 2^D \log^{1.1} \lambda \cdot \delta$-computationally indistinguishable.

6.2 Our Forest-Based PPPRF

Construction. We describe our forest-based PPPRF construction in Figure 4. The construction basically creates Z number of $\mathsf{PPPRFTree}$s, and uses the XOR of all of their results as the PRF evaluation outcome. To program the PRF at some point x, we select a random $\mathsf{PPPRFTree}$ such that the entire path to x is programmable, and program only that $\mathsf{PPPRFTree}$.

In Appendix C.2, we shall prove the following theorem.

Theorem 6.2 (Our forest-based PPPRF)**.** *Let $c > 0$, and let δ_0 be a function of λ, ℓ_{in}, and ℓ_{out} whose output range is $[0,1)$. Suppose we choose $\gamma = n^\epsilon$ for some constant $\epsilon \in (0,1)$, choose $\delta < \frac{1}{D \cdot 2^{D+1} \log^{1.1} \lambda} \cdot \delta_0$, and $Z > 2^{D+2} \cdot \log^{2.4} \lambda / \delta_0^2$ where $D = \lceil 1/\epsilon \rceil$. Then, our forest-based PPPRF construction satisfies δ_0-XOR functional programming privacy.*

Performance. If we use the parameters given in Theorem 6.2, we get a (forest-based) PPPRF with δ_0-XOR functional programming privacy with the following performance bounds where $\epsilon \in (0,1)$ is an arbitrarily small constant:

- *Key length.* The length of a normal key or a programmed key is $(\epsilon \cdot \lambda \cdot \log n)^{O(1/\epsilon)} \cdot \mathsf{poly} \log \lambda \cdot \ell_{\text{out}} / \delta_0^2 = (\epsilon \cdot \lambda \cdot \log n)^{O(1/\epsilon)} \cdot \ell_{\text{out}} / \delta_0^2$.
- *Time of* Prog, Eval, *and* PEval. The running time of these algorithms are bounded by $T_{\text{PRG}}(\lambda) \cdot (\epsilon \cdot \lambda \cdot \log n)^{O(1/\epsilon)} \cdot n^\epsilon \cdot \mathsf{poly} \log \lambda \cdot \ell_{\text{out}}^3 / \delta_0^4 = T_{\text{PRG}}(\lambda) \cdot (\epsilon \cdot \lambda \cdot \log n)^{O(1/\epsilon)} \cdot n^\epsilon \cdot \ell_{\text{out}}^3 / \delta_0^4$.

7 Preprocessing Private Information Retrieval

7.1 PIR Definitions

k-server pre-processing PIR. A k-server preprocessing PIR scheme is a protocol between a client and k servers both of which are *stateful*. The scheme consists of the following two sub-protocols:

1. $\mathsf{Preproc}(1^\lambda, \mathsf{DB})$: The pre-processing protocol is run only once at the beginning. The client receives 1^λ, while each server receives 1^λ and a database $\mathsf{DB} \in \{0,1\}^N$ as input. The client and servers then interact, and at the end of the protocol, the client and each server stores some state.

2. $\mathsf{Query}(x)$: The client receives an index x, it then interacts with the servers. At the end of the protocol, the client outputs an answer β. The $\mathsf{Query}(x)$ protocol can be repeated an aribitrary number of times after $\mathsf{Preproc}$ has been completed, and the client and all servers are stateful between the invocations.

Our forest-based PPPRF construction

Notation. Let $\mathsf{PPPRFTree}$ be the subroutine described in Figure 3. Let Z be the number of $\mathsf{PPPRFTrees}$.

$\underline{\mathsf{Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})}$:

- For $i \in [Z]$, let $\mathsf{msk}_i \leftarrow \mathsf{PPPRFTree.Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$.
- Output $\{\mathsf{msk}_1, \cdots, \mathsf{msk}_Z\}$.

$\underline{\mathsf{Eval}(\mathsf{msk} = \{\mathsf{msk}_1, \cdots, \mathsf{msk}_Z\}, x)}$: output $\bigoplus_{i=1}^{Z} \mathsf{PPPRFTree.Eval}(\mathsf{msk}_i, x)$.

$\underline{\mathsf{Prog}(\mathsf{msk} = \{\mathsf{msk}_1, \cdots, \mathsf{msk}_Z\}, x, v)}$:

- Choose a random $i \in [Z]$ subject to the constraint that msk_i is programmable for the entire path to x. Output $\perp$ if not found.
- Let $v' = v \oplus \mathsf{Eval}(\mathsf{msk}, x) \oplus \mathsf{PPPRFTree.Eval}(\mathsf{msk}_i, x)$.
- Let $\mathsf{sk}_i \leftarrow \mathsf{PPPRFTree.Prog}(\mathsf{msk}_i, x, v')$.
- Output the *unordered* set $\{\mathsf{msk}_1, \cdots, \mathsf{msk}_{i-1}, \mathsf{sk}_i, \mathsf{msk}_{i+1}, \cdots, \mathsf{msk}_Z\}$.

$\underline{\mathsf{PEval}(\mathsf{sk}, x)}$: output $\mathsf{Eval}(\mathsf{sk}, x)$.

Figure 4: Our forest-based PPPRF construction.

Assumption on state and number of rounds. Note that our definition is general: it admits both client-side preprocessing and server-side preprocessing. For the PIR schemes described in this paper, one can assume the following:

- the Query protocol involves a single round-trip, that is, the client sends a single message in parallel to each server, and it receives a single message in response from each server.
- each client's state may be updated after every query, whereas the server's state does not change between queries. In other words, one can imagine that the server stores some encoded version of the database after the preprocessing, and afterwards its state does not change.

Correctness. Correctness requires that for any n, Q which are polynomially bounded functions in λ, there exists a negligible function $\mathsf{negl}(\cdot)$, such that for any $\mathsf{DB}[0..N-1] \in \{0,1\}^N$, for any $x_1, \ldots, x_Q \in \{0, 1, \ldots, N-1\}$, the following experiment outputs 1 with probability at least $1 - \mathsf{negl}(\lambda)$:

- honestly execute the $\mathsf{Preproc}(1^\lambda, \mathsf{DB})$ protocol;
- for $i \in [Q]$, let β_i be the client's output in an honest executon of $\mathsf{Query}(x_i)$;
- finally, output 1 iff for all $i \in [Q]$, $\beta_i = \mathsf{DB}[x_i]$; else output 0.

Security. Given some function $\delta(\lambda, N)$, we say that a PIR scheme is δ-secure iff for any $j \in [k]$, there exists *stateless* PPT algorithms $\mathsf{QueryStateless}$ and Sim, such that for any N which is a polynomially bounded function in λ, for any index $x \in \{0, 1, \ldots, N-1\}$, $\mathsf{QueryStateless}(1^\lambda, x, N)$ is δ-computationally indistinguishable from $\mathsf{Sim}(1^\lambda, N)$, and moreover, for any non-uniform PPT

adversary $\mathcal{A}$ acting as the j-th server, its view in the following two experiments are computationally indistinguishable:

- $\mathsf{PIRReal}^{\mathcal{A}}(1^\lambda)$:
  1. $\mathcal{A}(1^\lambda)$ outputs DB of length N, run the $\mathsf{Preproc}(1^\lambda, N)$ protocol between the honest client and $\mathcal{A}$ acting as the j-th server;
  2. for $t = 1, 2, \ldots$: $\mathcal{A}$ outputs $x_t \in \{0, 1, \ldots, N-1\}$, run the $\mathsf{Query}(x_t)$ protocol between the honest client and $\mathcal{A}$ acting as the j-th server;
- $\mathsf{PIRIdeal}^{\mathcal{A}}(1^\lambda)$:
  1. $\mathcal{A}(1^\lambda)$ outputs DB of length N, run the $\mathsf{Preproc}(1^\lambda, N)$ protocol between the honest client and $\mathcal{A}$ acting as the j-th server;
  2. for $t = 1, 2, \ldots$: the adversary $\mathcal{A}$ outputs $x_t \in \{0, 1, \ldots, N-1\}$, send the message $\mathsf{QueryStateless}(1^\lambda, x_t, N)$ to $\mathcal{A}$.

In other words, we require that any individual PPT server cannot distinguish whether they are interacting with the real client or a stateless query algorithm $\mathsf{QueryStateless}$ which is given the query index; and further, the output of $\mathsf{QueryStateless}$ upon receiving the real queried index is δ-computationally indistinguishable from the output of a stateless simulator Sim which is not given the queried index. This captures the intuition that each query gives the adversary at most δ advantage in distinguishing which index is queried. For the special case where $\delta = 0$, we also say that the PIR scheme satisfies *full security*.

7.2 Weakly Secure PIR

We show how to use a δ-secure PPPRF scheme to construct a δ-secure preprocessing PIR scheme. In particular, we will describe a 1-server construction and a 2-server construction. Our constructions are inspired by Piano [ZPSZ24].

Privately puncturable pseudorandom set. First, we will use the PPPRF scheme of Section 6.2 to construct a privately programmable pseudorandom set scheme denoted PPPS — henceforth assume that N is an even power of 2:

- $\mathsf{Gen}(1^\lambda, N)$: let $\mathsf{msk}' \leftarrow \mathsf{PPPRF.Gen}(1^\lambda, \frac{1}{2}\log_2 N, \frac{1}{2}\log_2 N)$, choose a random offset $s \in \{0, 1, \ldots, \sqrt{N}-1\}$ and output $\mathsf{msk} := (\mathsf{msk}', s)$;
- $\mathsf{Set}(\mathsf{msk} = (\mathsf{msk}', s), N)$: for $i \in \{0, 1, \ldots, \sqrt{N}\}$, let $\Delta_i = (\mathsf{PPPRF.Eval}(\mathsf{msk}', i) + s) \mod \sqrt{N}$; and output the set $\{i \cdot \sqrt{N} + \Delta_i\}_{i \in \{0,1,\ldots,\sqrt{N}\}}$.
- $\mathsf{Member}(\mathsf{msk} = (\mathsf{msk}', s), N, x)$: to test if an index $x \in \{0, \ldots, N-1\}$ belongs to $\mathsf{PPPS.Set}(\mathsf{msk}, N)$, output 1 if $\mathsf{PPPRF.Eval}(\mathsf{msk}', \lfloor x/\sqrt{N} \rfloor) = (x - s) \mod \sqrt{N}$; else output 0.
- $\mathsf{Program}(\mathsf{msk} = (\mathsf{msk}', s), i, \Delta_i)$: to program chunk i's offset to Δ_i, let $\mathsf{sk}' \leftarrow \mathsf{PPPRF.Prog}(\mathsf{msk}', i, (\Delta_i - s) \mod \sqrt{N})$, and output the programmed key $\mathsf{sk} := (\mathsf{sk}', s)$.

Using the PPPRF scheme of Section 6.2, the programmed key sk has the same format as an unprogrammed key msk. Therefore, the same Set and Member algorithms can be run using a programmed key, too.

Intuitively, imagine that we have N indices numbered $0, 1, \ldots, N-1$. The above PPPS scheme can be used to sample a pseudorandom subset of size $\sqrt{N}$. Specifically, we will divide the N indices

1-server preprocessing PIR: supports $\sqrt{N}$ queries

Run $\mathsf{poly}\log\lambda$ parallel instances of the following scheme. For each query, if any instance does not return $\perp$, output that answer.

$\underline{\mathsf{Preproc}(1^\lambda, \mathsf{DB})}$:

- Client generates a pseudorandom permutation (PRP) key and sends it to the server. Henceforth, we assume the DB has been permuted using this PRP.
- Let $\mathsf{len} := O(\sqrt{N})$, and for $j \in [\mathsf{len}]$: sample $\mathsf{msk}_j \leftarrow \mathsf{PPPS.Gen}(1^\lambda, N)$.
- For each chunk $i \in \{0, 1, \ldots, \sqrt{N}-1\}$, sample $\mathsf{poly}\log\lambda$ random indices that belong to the chunk. Let R_i be the set of indices for chunk i.
- Make a single streaming pass over the database:
  - *Hint table*: for all $j \in [\mathsf{len}]$, compute and store $p_z := \oplus_{z \in \mathsf{Set}(\mathsf{msk}_i, N)} \mathsf{DB}[z]$.
  - *Replacement entries*: for each chunk $i \in \{0, 1, \ldots, \sqrt{N}-1\}$, each $y \in R_i$, store $p_y := \mathsf{DB}[y]$.
- Store the hint table $\{\mathsf{msk}_j, p_j\}_{j \in [\mathsf{len}]}$, and the replacement entries $\{y, p_y\}_{y \in R_0 \cup \ldots \cup R_{\sqrt{N}-1}}$.

$\underline{\mathsf{Query}(x)}$:

Client:

1. If x has been queried among the past up to $\sqrt{N}$ queries, output the same answer as before and execute $\mathsf{Query}(r)$ instead for a random index r that has not been queried (but ignore its output). Else, continue with the following.
2. Find a random hint (msk^*, p^*) in the hint table such that $\mathsf{PPPS.Member}(\mathsf{msk}^*, x) = 1$. Find the next unconsumed replacement entry (y, p_y) that belongs to the same chunk as x. If either is not found, sample a random msk^* such that $\mathsf{PPPS.Member}(\mathsf{msk}^*, x) = 1$, let $p^* = \perp$, let y be a random index within the same chunk as x, and let $p_y := 0$.
3. Let $\mathsf{sk} \leftarrow \mathsf{PPPS.Program}(\mathsf{msk}^*, \lfloor x/\sqrt{N} \rfloor, y \mod \sqrt{N})$.
4. Send sk to server, and receive a response p.
5. Sample $\mathsf{msk}' \leftarrow \mathsf{PPPS.Gen}(1^\lambda, N)$ subject to $\mathsf{Member}(\mathsf{msk}', x) = 1$. Replace the consumed hint with the broken hint $(\mathsf{msk}', \perp)$.
6. If $p^* = \perp$, i.e., the hint is broken or the earlier step 2 did not find a match, then output $\perp$; else output $p \oplus p^* \oplus p_y$.

Server: upon receiving msk, return $\oplus_{z \in \mathsf{PPPS.Set}(\mathsf{msk}, N)} \mathsf{DB}[z]$.

Figure 5: Single-server preprocessing PIR for bounded queries. The scheme satisfies δ-security if the underlying PPPRF satisfies δ-XOR functional programming privacy. We can upgrade the scheme to support unbounded queries as in [ZPSZ24].

into $\sqrt{N}$ continguous chunks and each chunk is $\sqrt{N}$ in size. When we sample a secret key msk, the set generated by msk includes one index per chunk. Specifically, $\Delta_i := (\mathsf{PPPRF.Eval}(\mathsf{msk}, i) + s) \mod \sqrt{N}$ gives the offset within the i-th chunk where $i \in \{0, 1, \ldots, \sqrt{N}-1\}$, i.e., the index $i \cdot \sqrt{N} + \Delta_i$ is contained in the set generated by msk.

7.2.1 1-Server Scheme

In Figure 5, we present a 1-server preprocessing PIR scheme that supports up to $\sqrt{N}$ queries. We can use the same pipelining idea described in Zhou et al. [ZPSZ24] to extend the scheme to support unbounded number of queries. The idea is that after the initial preprocessing that is done, in every window of $\sqrt{N}$ queries, we run the preprocessing for the next window of $\sqrt{N}$ queries, and the work is spread across each query. We can use known small-domain PRP constructions [HMR12] to instantiate the PRP used to permute the database upfront. Specifically, each PRP evaluation takes only $\widetilde{O}_\lambda(1)$ time.

Performance. We now analyze the performance of our 1-server preprocessing PIR. Our analysis accounts for the cost of all $\mathsf{poly}\log\lambda$ parallel instances. Henceforth, let $L_{\text{PPPRF}} := L_{\text{PPPRF}}(\lambda, \ell_{\text{in}}, \ell_{\text{out}}, \delta)$ denote an upper bound on the length of a normal and programmed key and $T_{\text{PPPRF}} := T_{\text{PPPRF}}(\lambda, \ell_{\text{in}}, \ell_{\text{out}}, \delta)$ denote an upper bound on the evaluation and programming cost of the underlying PPPRF.

- *Space.* The client space, which is dominated by the cost of storing $\sqrt{N} \cdot \mathsf{poly}\log\lambda$ keys, is bounded by $\sqrt{N} \cdot \mathsf{poly}\log\lambda \cdot L_{\text{PPPRF}}$. The server only needs to store the database itself and thus the server space is N.
- *Preprocessing cost.* During the preprocessing, the client downloads one chunk from the server at a time, and updates all len cumulative parities in each of the $\mathsf{poly}\log\lambda$ instances. In other words, the client can compute the hint table and replacement entries for all $\mathsf{poly}\log\lambda$ instances in a single scan. Therefore, the client computation during preprocessing is bounded by $\mathsf{poly}\log\lambda \cdot N \cdot T_{\text{PPPRF}}$. The preprocessing communication and server computation is N.
- *Query cost.*
  - *Online cost.* The online cost is on the critical path of getting the answer. The online communication cost is $\mathsf{poly}\log\lambda \cdot L_{\text{PPPRF}}$. The online server computation is bounded by $\mathsf{poly}\log\lambda \cdot \sqrt{N} \cdot T_{\text{PPPRF}}$. The online client computation is bounded by $\mathsf{poly}\log\lambda \cdot \sqrt{N} \cdot T_{\text{PPPRF}}$ with $1 - \mathsf{negl}(\lambda)$ probability. Note that the $\mathsf{negl}(\lambda)$ failure is due to the negligible probability that the rejection sampling (to find a key $\mathsf{msk}' \leftarrow \mathsf{PPPS.Gen}$ subject to containing the present query) exceeds the above stated time bound.
  - *Offline cost.* The offline cost is the cost of performing the next phase's preprocessing amortized across the present phase of $\sqrt{N}$ queries. The offline bandwidth and server computation is $\sqrt{N}$ per query. The offline client computation is $\mathsf{poly}\log\lambda \cdot \sqrt{N} \cdot T_{\text{PPPRF}}$ per query.

For the special case when $\delta = 1/\mathsf{poly}\log\lambda$ which is what we need in the security amplification later, and using the PPPRF of Section 6.2, we have $L_{\text{PPPRF}} = (\epsilon\lambda\log N)^{O(1/\epsilon)}$, and $T_{\text{PPPRF}} = (\epsilon\lambda\log N)^{O(1/\epsilon)} \cdot N^\epsilon \cdot T_{\text{PRG}}(\lambda)$ for an arbitrary constant $\epsilon \in (0,1)$. In this case, we have the following performance:

- *Space:* client space $= \sqrt{N} \cdot (\epsilon\lambda\log N)^{O(1/\epsilon)}$, server space $= N$;
- *Preprocessing:* client computation $= N^{1+\epsilon} \cdot (\epsilon\lambda\log N)^{O(1/\epsilon)} \cdot T_{\text{PRG}}(\lambda)$, bandwidth $= N$, server computation $= N$;
- *Per-query cost:* online bandwidth $= (\epsilon\lambda\log N)^{O(1/\epsilon)}$, online client/server computation $= N^{\frac{1}{2}+\epsilon}(\epsilon\lambda\log N)^{O(1/\epsilon)} \cdot T_{\text{PRG}}(\lambda)$; offline bandwidth/server computation $= \sqrt{N}$, offline client computation $= N^{\frac{1}{2}+\epsilon}(\epsilon\lambda\log N)^{O(1/\epsilon)} \cdot T_{\text{PRG}}(\lambda)$.

Theorem 7.1 (δ-secure 1-server preprocessing PIR)**.** *Suppose that the underlying* PPPRF *satisfies programming correctness and δ-XOR functional programming privacy. Further, suppose that the length of the hint table* $\mathsf{len} = C\sqrt{N}$ *for a suitably large constant $C > 1$. Then, the 1-server preprocessing PIR scheme in Figure 5 satisfies correctness and δ-security.*

We defer the proof to Appendix D.

7.2.2 2-Server Scheme

The 2-server scheme can be obtained fairly easy by modifying the 1-server scheme. The client directly downloads the preprocessed hint from the *left* server, and interacts with the *right* server during the query phase. This saves the client from having to download the whole database during preprocessing. We defer the detailed description and the theorems to Appendix D.2.

8 Security Amplification

Section 7.2 gives δ-secure preprocessing PIR schemes for the single-server and two-server settings. In this section, we describe how to amplify the security to achieve negligible failure probability against any polynomial-time adversary.

8.1 Additional Preliminaries

Locally decodable code. A locally decodable code (LDC) allows us to encode a string henceforth denoted DB of N bits into a codeword $\widetilde{\mathsf{DB}}$, such that to query $\mathsf{DB}[i]$, it suffices to query Q locations in the codeword $\widetilde{\mathsf{DB}}$. Moreover, we require that the Q locations queried satisfy σ-wise independence. Standard definitions of additionally require error correction, but for our application, we will not need the error correction property, so we omit it from the definition. Formally, an LDC over the field $\mathbb{F}_p$ has the following algorithms:

- $\widetilde{\mathsf{DB}} \leftarrow \mathsf{Encode}(\mathsf{DB})$: takes in a $\mathsf{DB} \in \mathbb{F}_p^N$ outputs a codeword $\widetilde{\mathsf{DB}} \in \mathbb{F}_p^{\widetilde{N}}$.
- $i_1, \cdots, i_Q \leftarrow \mathsf{Query}(i)$: outputs Q queries to the encoded database for an original query $i \in [N]$.
- $w \leftarrow \mathsf{Decode}(i_1, \cdots, i_Q, w_1, \cdots, w_Q)$: takes in Q indices $i_1, \cdots, i_Q$ and Q values $w_1, \cdots, w_Q \in \mathbb{F}_p$ and outputs the reconstructed value w.

Correctness requires that for any $\mathsf{DB} \in \mathbb{F}_q^N$, any $i \in \{0, 1, \ldots, N-1\}$, the following holds with probability 1: let $\widetilde{\mathsf{DB}} \leftarrow \mathsf{Encode}(\mathsf{DB})$, let $i_1, \ldots, i_Q \leftarrow \mathsf{Query}(i)$, then $\mathsf{Decode}(i_1, \ldots, i_Q, \widetilde{\mathsf{DB}}[i_1], \ldots, \widetilde{\mathsf{DB}}[i_Q]) = \mathsf{DB}[i]$.

An LDC is said to satisfy *σ-smoothness*, iff for any $i \in \{0, 1, \ldots, N-1\}$, any σ out of the Q queries output by $\mathsf{Query}(i)$ are jointly distributed as σ independent and uniform random indices from $\{0, 1, \ldots, \widetilde{N} - 1\}$.

Theorem 8.1 (Reed-Muller code [HOWW19])**.** *For any constant $\epsilon > 0$, there exists a σ-smooth LDC such that the field size $p = (\sigma \log N)^{O(1/\epsilon)}$, the codeword length $\widetilde{N} = N^{1+\epsilon} \cdot (\sigma \log N)^{O(1/\epsilon)}$, and the number of queries $Q = (\sigma \log N)^{O(1/\epsilon)}$. Further, the encoding time is $\widetilde{O}(\widetilde{N})$ and the decoding time is $\widetilde{O}(Q)$ where $\widetilde{O}(\cdot)$ hides* $\mathsf{poly}\log(N, 1/\epsilon, \sigma)$ *factors.*

Security Amplification for k-server preprocessing PIR

$\underline{\mathsf{Preproc}(1^\lambda, \mathsf{DB})}$:

- Each server computes $\widetilde{\mathsf{DB}} \leftarrow \mathsf{LDC.Encode}(\mathsf{DB})$.
- The client and the k servers run $\mathsf{PIR.Preproc}(1^\lambda, \widetilde{\mathsf{DB}})$.

$\underline{\mathsf{Query}(x)}$: Client computes $x_1, \cdots, x_Q \leftarrow \mathsf{LDC.Query}(x)$, and for $j = 1$ to Q:

- for each server $i \in [k]$, the client sends to the i-th server[a]:

$$\mathsf{Set}(\mathsf{PIR.Query}_i(x_j), \underbrace{\mathsf{PIR.Sim}_i(1^\lambda, N), \ldots, \mathsf{PIR.Sim}_i(1^\lambda, N)}_{d-1})$$

  where $\mathsf{PIR.Query}_i$ and $\mathsf{PIR.Sim}_i$ denote the output of the query algorithm for the i-th server, and the simulator for the i-th server of the underlying PIR.
- each server: upon receiving a set of d queries, answer all queries in the set.
- client sets w_j as the answer to the sole real query and ignores the others.

Finally, the client outputs $\mathsf{LDC.Decode}(x_1, \cdots, x_Q, w_1, \cdots, w_Q)$.

[a] To send a set, simply send a random permutation of its elements.

Figure 6: Security amplification for k-server preprocessing PIR. Here, PIR denotes a k-server preprocessing PIR scheme and LDC denotes a locally decodable code.

8.2 Security Amplification of PIR

Given a δ-secure k-server preprocessing PIR scheme, we show how to amplify the security such that the security failure probability becomes negligibly small for any polynomial-time adversary. The construction is shown in Theorem 8.2 with the following choice of parameters.

Parameter choices. We will use Theorem 8.1 to instantiate the underlying LDC. Fix an arbitrarily small constant ϵ. We will choose the following parameters. Let $\sigma = \log^2 \lambda$. Let $Q = (\sigma \log N)^{\Theta(1/\epsilon)}$. Choose $\delta < \sigma/6Q$ and $d > \log^2 \lambda/\delta^2$. It is not hard to see that both Q and d are polylogarithmically bounded.

Theorem 8.2 (PIR security amplification)**.** *Suppose we choose the parameters as above. Suppose that the underlying* PIR *scheme satisfies correctness and δ-security, and that we use the* LDC *of Theorem 8.1 with message length N, query length Q, and smoothness parameter σ. Then, the construction Figure 6 is a k-server preprocessing PIR scheme that satisfies correctness and full security.*

Our construction requires an underlying PIR scheme with an $N^{1+\epsilon} \cdot \log^{O(1/\epsilon)}(\lambda, N)$-sized database, and for each query, the client makes $Q \cdot d = \log^{O(1/\epsilon)}(\lambda, N)$ queries to the underlying PIR or the PIR's simulator algorithm. Using the above parameters and plugging in the 1-server and 2-server schemes of Section 7.2, we get the following corollaries — specifically, we can split ϵ, and use the parameter $\epsilon/2$ for the underlying PIR and $\epsilon/2$ for the amplification protocol:

Corollary 8.3 (Fully secure 1-server preprocessing PIR)**.** *Assume the existence of one-way functions, and let $\epsilon \in (0,1)$ be an arbitrary constant. Then, there exists a fully secure 1-server preprocessing PIR scheme that achieves the following performance:*

- *Space: client space $= N^{\frac{1}{2}+\epsilon} \cdot (\epsilon\lambda \log N)^{O(1/\epsilon)}$, server space $= N^{1+\epsilon} \cdot \log^{O(1/\epsilon)}(N, \lambda)$;*
- *Preprocessing: client computation $= N^{1+\epsilon} \cdot (\epsilon\lambda \log N)^{O(1/\epsilon)} \cdot T_{\mathrm{PRG}}(\lambda)$, bandwidth = server computation $= N^{1+\epsilon} \cdot \log^{O(1/\epsilon)}(N, \lambda)$;*
- *Per-query cost: online bandwidth $= (\epsilon\lambda \log N)^{O(1/\epsilon)}$, online client/server computation $= N^{\frac{1}{2}+\epsilon}(\epsilon\lambda \log N)^{O(1/\epsilon)} \cdot T_{\mathrm{PRG}}(\lambda)$; offline bandwidth/server computation $= \sqrt{N} \cdot \log^{O(1/\epsilon)}(N, \lambda)$, offline client computation $= N^{\frac{1}{2}+\epsilon}(\epsilon\lambda \log N)^{O(1/\epsilon)} \cdot T_{\mathrm{PRG}}(\lambda)$.*

Corollary 8.4 (Fully secure 2-server preprocessing PIR)**.** *Assume the existence of one-way functions, and let $\epsilon \in (0,1)$ be an arbitrary constant. Then, there exists a fully secure 2-server preprocessing PIR scheme that achieves the following performance:*

- *Space: client space $= N^{\frac{1}{2}+\epsilon} \cdot (\epsilon\lambda \log N)^{O(1/\epsilon)}$, server space $= N^{1+\epsilon} \cdot \log^{O(1/\epsilon)}(N, \lambda)$;*
- *Preprocessing: client computation/bandwidth $= N^{\frac{1}{2}+\epsilon} \cdot (\epsilon\lambda \log N)^{O(1/\epsilon)}$, left server computation $= N^{1+\epsilon}(\epsilon\lambda \log N)^{O(1/\epsilon)} \cdot T_{\mathrm{PRG}}(\lambda)$, right server computation = 0;*
- *Per-query cost (both online and offline): bandwidth $= (\epsilon\lambda \log N)^{O(1/\epsilon)}$, client/server computation $= N^{\frac{1}{2}+\epsilon}(\epsilon\lambda \log N)^{O(1/\epsilon)} \cdot T_{\mathrm{PRG}}(\lambda)$.*

Acknowledgments

This work is in part supported by a grant from ONR, a grant from the DARPA SIEVE program under a subcontract from SRI, a gift from Cisco, Samsung MSL, NSF awards under grant numbers 1705007, 2128519 and 2044679.

References

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *S&P*, 2018.

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *Proceedings of the 17th International Conference on Public-Key Cryptography — PKC 2014 - Volume 8383*, page 501–519, Berlin, Heidelberg, 2014. Springer-Verlag.

[BGIK22] Elette Boyle, Niv Gilboa, Yuval Ishai, and Victor I. Kolobov. Programmable distributed point functions. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 121–151, Cham, 2022. Springer Nature Switzerland.

[BIM00] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: Pir with preprocessing. In *CRYPTO*, pages 55–73, 2000.

[BKM17] Dan Boneh, Sam Kim, and Hart William Montgomery. Private puncturable PRFs from standard lattice assumptions. In *EUROCRYPT*, pages 415–445, 2017.

[BLW17] Dan Boneh, Kevin Lewi, and David J. Wu. Constraining pseudorandom functions privately. In *PKC*, 2017.

[BTVW17] Zvika Brakerski, Rotem Tsabary, Vinod Vaikuntanathan, and Hoeteck Wee. Private constrained prfs (and more) from LWE. In *TCC*, 2017.

[BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II*, volume 8270 of *Lecture Notes in Computer Science*, pages 280–300. Springer, 2013.

[CC17] Ran Canetti and Yilei Chen. Constraint-hiding constrained PRFs for NC^1 from LWE. In *EUROCRYPT*, pages 446–476, 2017.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*, 1995.

[CHK22] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In *Eurocrypt*, 2022.

[CK20] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *EUROCRYPT*, 2020.

[CMS99] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT*, pages 402–414, 1999.

[Con] Graeme Connell. Technology deep dive: Building a faster oram layer for enclaves. https://signal.org/blog/building-faster-oram/.

[DCMO00] Giovanni Di Crescenzo, Tal Malkin, and Rafail Ostrovsky. Single database private information retrieval implies oblivious transfer. In *EUROCRYPT*, 2000.

[DG16] Zeev Dvir and Sivakanth Gopi. 2-server pir with subpolynomial communication. *J. ACM*, 63(4), 2016.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4), August 1986.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[Gol87] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.

[GZS24] Ashrujit Ghoshal, Mingxun Zhou, and Elaine Shi. Efficient pre-processing pir without public-key cryptography. In *Advances in Cryptology – EUROCRYPT 2024: 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26–30, 2024, Proceedings, Part VI*, page 210–240, 2024.

[HDCG+23] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, , and Nickolai Zeldovich. Private web search with Tiptoe. In *29th ACM Symposium on Operating Systems Principles (SOSP)*, Koblenz, Germany, October 2023.

[HHCG+22] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. Cryptology ePrint Archive, Paper 2022/949, 2022. https://eprint.iacr.org/2022/949.

[HMR12] Viet Tung Hoang, Ben Morris, and Phillip Rogaway. An enciphering scheme based on a card shuffle. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2012.

[HOWW19] Ariel Hamlin, Rafail Ostrovsky, Mor Weiss, and Daniel Wichs. Private anonymous data access. In *EUROCRYPT*, 2019.

[HPPY24] Alexander Hoover, Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Plinko: Single-server PIR with efficient updates via invertible prfs. *IACR Cryptol. ePrint Arch.*, page 318, 2024.

[ISW24] Yuval Ishai, Elaine Shi, and Daniel Wichs. PIR with client-side preprocessing: Information-theoretic constructions and lower bounds. In *CRYPTO*, 2024.

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

[Lip05] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In *Information Security*, pages 314–328, 2005.

[Lip09] Helger Lipmaa. First CPIR protocol with data-dependent computation. In *ICISC*, 2009.

[LMW23] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic ram computation from ring lwe. In *STOC*, 2023.

[LP22] Arthur Lazzaretti and Charalampos Papamanthou. Single server pir with sublinear amortized time and polylogarithmic bandwidth. Cryptology ePrint Archive, Paper 2022/830, 2022. https://eprint.iacr.org/2022/830.

[LP23] Arthur Lazzaretti and Charalampos Papamanthou. Treepir: Sublinear-time and polylog-bandwidth private information retrieval from ddh. In *CRYPTO*, 2023.

[MIR23] Muhammad Haris Mughees, Sun I, and Ling Ren. Simple and practical amortized sublinear private information retrieval. Cryptology ePrint Archive, Paper 2023/1072, 2023.

[MT10] Ueli M. Maurer and Stefano Tessaro. A hardcore lemma for computational indistinguishability: Security amplification for arbitrarily weak prgs with optimal stretch. In *TCC 2010*, volume 5978, pages 237–254, 2010.

[MW22] Samir Jordan Menon and David J. Wu. SPIRAL: Fast, high-rate single-server PIR via FHE composition. In *IEEE S&P*, 2022.

[pri] Improving dns privacy with oblivious doh in 1.1.1.1. https://blog.cloudflare.com/oblivious-dns/.

[PS18] Chris Peikert and Sina Shiehian. Privately constraining and programming PRFs, the LWE way. In *Public Key Cryptography (2)*, volume 10770 of *Lecture Notes in Computer Science*, pages 675–701. Springer, 2018.

[SACM21] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce Maggs. Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. In *CRYPTO*, 2021.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.

[WY05] David Woodruff and Sergey Yekhanin. A geometric approach to information-theoretic private information retrieval. In *20th Annual IEEE Conference on Computational Complexity (CCC'05)*, pages 275–284, 2005.

[ZLTS23] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. Optimal single-server private information retrieval. In *EUROCRYPT*, 2023.

[ZPSZ24] Mingxun Zhou, Andrew Park, Elaine Shi, and Wenting Zheng. Piano: Extremely simple, single-server pir with sublinear server computation. In *IEEE S& P*, 2024.

Supplementary Materials

A Additional Related Work

We now review additional related work. Computationally secure single-server (classical) PIR with $\widetilde{O}(1)$ bandwidth and linear computation per query can be built from various assumptions such as LWE, ϕ-hiding, decisional composite residuosity (DCR), and linear homomorphic encryption [HHCG+22, MW22, ACLS18, CMS99, Lip09, Lip05]. The recent ground-breaking work of Lin, Mook, and Wichs [LMW23] showed a PIR scheme with server-side preprocessing that achieves polylogarithmic bandwidth and computation per query, consuming $N^{1+\epsilon}$ server space. All of these works require cryptographic primitives that imply public-key cryptography, and this is inherent without client-side preprocessing [DCMO00]. If we are not concerned about having (super-)linear computation per query, then it is known how to achieve $N^{o(1)}$ bandwidth in the information-theoretic setting due to the elegant work of Dvir and Gopi [DG16].

Some earlier works also considered preprocessing PIR but in the single-query setting (e.g., Theorem 11 of Corrigan-Gibbs and Kogan [CK20]). With such schemes, a separate preprocessing is needed for every query, so the total cost per query is still high. In our work, we focus on preprocessing PIR that supports an unbounded number of queries, so we can amortize the preprocessing cost over the unbounded number of queries.

The line of work on Oblivious RAM (ORAM) [GO96, Gol87, SCSL11] is also related in the sense that ORAM also provides access pattern privacy. However, with ORAM, the server needs to store a separate copy of the (encrypted and ORAM'ed) database for each client. In some deployment

scenarios such as Signal's private contact discovery application [Con], they use a secure processor on the cloud server, such that the secure processor acts as a globally unique client. This avoids the multiplicative blowup in server space related to the number of clients.

B Deferred Proofs for Section 5

B.1 Proofs of Theorem 5.1 and Theorem 5.2

The proof of Boyle et al. [BGIK22] implies the following technical lemma.

Lemma B.1 (Technical lemma imported from Boyle et al. [BGIK22])**.** *Suppose that $m = \omega(n \cdot \log \lambda)$. Let* pPRF *denote a secure puncturable PRF scheme with input domain $\{0, \ldots, m-1\}$ and output range $\{0, \ldots, n-1\}$. Then, for any $x \in \{0, \ldots, n-1\}$, the following probability ensembles are $\sqrt{n/m}$-computationally indistinguishable:*

- **Real***: Sample $s \xleftarrow{\$} \{0, \ldots, n-1\}$, $\mathsf{sk} \leftarrow \mathsf{pPRF.Gen}(1^\lambda)$, and choose a random r from $\{0, \ldots, m-1\}$ subject to $\mathsf{pPRF.Eval}(\mathsf{sk}, r) + s = x \mod n$; let $\mathsf{sk}' \leftarrow \mathsf{pPRF.Punct}(\mathsf{sk}, r)$, and output (sk', s) or $\perp$ if no such r is found.*
- **Ideal***: Sample $s \xleftarrow{\$} \{0, \ldots, n-1\}$, $\mathsf{sk} \leftarrow \mathsf{pPRF.Gen}(1^\lambda)$, and choose a random $r \xleftarrow{\$} \{0, \ldots, m-1\}$; let $\mathsf{sk}' \leftarrow \mathsf{pPRF.Punct}(\mathsf{sk}, r)$, and output (sk', s).*

Proof of Theorem 5.1. We now prove Theorem 5.1.

- *Correctness.* Correctness is guaranteed as long as the Prog procedure does not output $\perp$. Imagine that we are throwing m balls $i \in \{0, 1, \ldots, m-1\}$ into n bins, based on the outcome $(\mathsf{pPRF}(\mathsf{psk}, i) + s) \mod n$. Since the underlying pPRF satisfies pseudorandomness and by the Chernoff bound, the probability that some bin is empty is upper bounded by $\mathsf{negl}(\lambda)$. Thus, correctness holds with probability $1 - \mathsf{negl}(\lambda)$.
- *Pseudorandomness.* The proof of pseudorandomness is straightforward since the evaluation outcome is XOR'ed with the pseudorandom padding $\mathsf{xPRF}(\mathsf{xsk}, x)$.
- *Functional programming privacy.* In the functional programming privacy security experiment, the adversary can choose either f^0 or f^1, where $f^b(v) = v \oplus b$. Henceforth, suppose that the adversary chooses f^0 since the proof is similar to the case of f^1.

  We can define the simulator Sim to sample sk' and s just like in the **Ideal** game of Theorem B.1. Further, Sim additionally samples a random xPRF key xsk, and outputs $(\mathsf{sk}', \mathsf{xsk}, s)$. We now show that the real and ideal experiments RealProgPriv and IdealProgPriv are $\sqrt{n/m}$-computationally indistinguishable. In the real experiment RealProgPriv, the adversary $\mathcal{A}$ receives the programmed key $\mathsf{sk}_{\text{real}} = (\mathsf{pPRF.Punct}(\mathsf{sk}, r), \mathsf{xsk}, s)$ where r is chosen at random subject to $\mathsf{pPRF.Eval}(\mathsf{sk}, r) + s = n - 1 \mod n$. In the ideal experiment IdealProgPriv, adversary $\mathcal{A}$ receives the programmed key $\mathsf{sk}_{\text{ideal}} = (\mathsf{pPRF.Punct}(\mathsf{sk}, r), \mathsf{xsk}, s)$ where r is chosen at random. By Theorem B.1 and the post-processing lemma, the distributions $\mathsf{sk}_{\text{real}}$ and $\mathsf{sk}_{\text{ideal}}$ are $\sqrt{n/m}$-computationally indistinguishable for an arbitrary x submitted by the adversary. Thus, we conclude that RealProgPriv and IdealProgPriv are $\sqrt{n/m}$-computationally indistinguishable for any non-uniform PPT $\mathcal{A}$.

Proof of Theorem 5.2. Next, we prove Theorem 5.2. Recall that Goldreich, Goldwasser, and Micali construction [GGM86] is a tree-based PRF construction based on a pseudorandom generator (PRG). Given a random seed $s \in \{0,1\}^\lambda$ at the root, we can expand the seed with the PRG and compute a random λ-bit key for each of the children. This process continues until we compute a key for each node in the tree. The tree has m leaves, and the leaf level is a little special: for our setting, the key at the leaf level has bit length $\log_2 n$ (as opppsed to λ bits). Now, the resulting key at each leaf $i \in \{0, 1, \ldots, m-1\}$ is the PRF's evaluation at the point i.

A key sk_{-r} punctured at the point $r \in \{0, 1, \ldots, m-1\}$ consists of

- the point r, and
- $L = O(\log_2 m)$ keys denoted $\{\mathsf{sk}_1, \ldots, \mathsf{sk}_L\}$: specifically, the keys associated with all nodes that are sibling to the path from leaf r to the root. We use sk_i to denote the key at level i of the tree, where the root is assumed to be at level 0.

Claim B.2. For any fixed r, sk_{-r} is computationally indistinguishable from r along with L random strings of appropriate lengths.

The above claim can be proven through a standard hybrid argument assuming the pseudorandomness of the PRG. In particular, let Hyb_0 be the real distribution of sk_{-r}, and let Hyb_i be the following distribution: sample $\mathsf{sk}_1, \ldots, \mathsf{sk}_i$ at random; sample sk'_i at random, and compute $\mathsf{sk}_{i+1}, \ldots, \mathsf{sk}_L$ honestly assuming sk'_i is associated with the root of the subtree that contains the keys $\mathsf{sk}_{i+1}, \ldots, \mathsf{sk}_L$; and output $(r, \mathsf{sk}_1, \ldots, \mathsf{sk}_L)$. Clearly, Hyb_L is the same as the distribution specified in Claim B.2. The pseudorandomness of PRG guarantees that any two adjacent pair of hybrids are computationally indistinguishable through a straightforward reduction.

Given Claim B.2, Theorem 5.2 follows by the post-processing lemma of computational indistinguishability.

PRF with general functional programming privacy

Notation: PPPRF is a PRF with δ-XOR functional programming privacy, and 1-bit outcome

Assume: the input domain is $\{0, 1, \ldots, n-2\}$ where $n = 2^{\ell_{\text{in}}}$

$\underline{\mathsf{Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})}$:

- For $i \in [\ell_{\text{out}}]$, let $\mathsf{msk}_i \leftarrow \mathsf{PPPRF.Gen}(1^\lambda, \ell_{\text{in}}, 1)$, and let $\mathsf{pad} \xleftarrow{\$} \{0,1\}^{\ell_{\text{out}}}$.
- Output $\mathsf{msk} = (\mathsf{msk}_1, \cdots, \mathsf{msk}_{\ell_{\text{out}}}, \mathsf{pad})$.

$\underline{\mathsf{Eval}(\mathsf{msk} = (\mathsf{msk}_1, \cdots, \mathsf{msk}_{\ell_{\text{out}}}, \mathsf{pad}), x)}$:

- Output $(\mathsf{PPPRF.Eval}(\mathsf{msk}_1, x) \| \cdots \| \mathsf{PPPRF.Eval}(\mathsf{msk}_{\ell_{\text{out}}}, x)) \oplus \mathsf{pad}$.

$\underline{\mathsf{Prog}(\mathsf{msk} = (\mathsf{msk}_1, \cdots, \mathsf{msk}_{\ell_{\text{out}}}, \mathsf{pad}), x, v)}$:

- For each $i \in [\ell_{\text{out}}]$, let $\mathsf{sk}_i \leftarrow \mathsf{PPPRF.Prog}(\mathsf{msk}_i, x, v_i \oplus \mathsf{pad}_i)$ where v_i and pad_i denote the i-th bit of v and pad, respectively.
- Output $\mathsf{sk} = (\mathsf{sk}_1, \cdots, \mathsf{sk}_{\ell_{\text{out}}}, \mathsf{pad})$.

$\underline{\mathsf{PEval}(\mathsf{sk} = (\mathsf{sk}_1, \cdots, \mathsf{sk}_{\ell_{\text{out}}}, \mathsf{pad}), x)}$:

- Output $(\mathsf{PPPRF.PEval}(\mathsf{sk}_1, x) \| \cdots \| \mathsf{PPPRF.PEval}(\mathsf{sk}_{\ell_{\text{out}}}, x)) \oplus \mathsf{pad}$.

Figure 7: Small-domain PRF with general functional programming privacy.

B.2 Proofs of Theorem 5.3 and Theorem 5.4

The performance bounds and correctness are easy to see. We focus on proving functional programming privacy. Suppose that the function family $\mathcal{F}$ is δ'-random-to-random. We want to show that $\mathsf{RealProgPriv}$ is $(\delta' + \ell_{\text{out}} \cdot \delta)$-indistinguishable from $\mathsf{IdealProgPriv}$. We prove this through a sequence of hybrid experiments.

Experiment Hyb_1. We start with the following experiment Hyb_1, which is identical to $\mathsf{RealProgPriv}$ when we plug in our construction in Figure 7.

Experiment Hyb_1

Let $x, f \leftarrow \mathcal{A}$
Sample $\mathsf{msk}_1, \cdots, \mathsf{msk}_{\ell_{\text{out}}} \leftarrow \mathsf{PPPRF.Gen}(1^\lambda, \ell_{\text{in}}, 1)$, $\mathsf{pad} \xleftarrow{\$} \{0,1\}^{\ell_{\text{out}}}$
Let $v \leftarrow (\mathsf{PPPRF.Eval}(\mathsf{msk}_1, x) \| \cdots \| \mathsf{PPPRF.Eval}(\mathsf{msk}_{\ell_{\text{out}}}, x)) \oplus \mathsf{pad}$
Let $v' \leftarrow f(v)$
For each $i \in [\ell_{\text{out}}]$, let $\mathsf{sk}_i \leftarrow \mathsf{PPPRF.Prog}(\mathsf{msk}_i, x, v'_i \oplus \mathsf{pad}_i)$
Let $\mathsf{sk} = (\mathsf{sk}_1, \ldots, \mathsf{sk}_{\ell\text{out}}, \mathsf{pad})$
Output $\mathcal{A}(\mathsf{sk})$

Experiment Hyb_2. Hyb_2 is a rewrite of Hyb_1 by switching the sampling order. Recall that in Hyb_1, we sample pad and msk, and then we compute $v \leftarrow (\mathsf{PPPRF.Eval}(\mathsf{msk}_1, x) \| \cdots \| \mathsf{PPPRF.Eval}(\mathsf{msk}_{\ell_{\text{out}}}, x)) \oplus \mathsf{pad}$ which is uniformly distributed. In Hyb_2, we sample v and msk first and then we compute pad — see below. Thus, Hyb_2 is identically distributed as Hyb_1.

Experiment Hyb_2

Let $x, f \leftarrow \mathcal{A}$
Let $v \xleftarrow{\$} \{0,1\}^{\ell_{\text{out}}}$
Let $v' \leftarrow f(v)$
Sample $\mathsf{msk}_1, \cdots, \mathsf{msk}_{\ell_{\text{out}}} \leftarrow \mathsf{PPPRF.Gen}(1^\lambda, \ell_{\text{in}}, 1)$
For each $1 \le i \le \ell_{\text{out}}$, let $\mathsf{sk}_i \leftarrow \mathsf{PPPRF.Prog}(\mathsf{msk}_i, x, \mathsf{PPPRF.Eval}(\mathsf{msk}_i, x) \oplus v_i \oplus v'_i)$
$\mathsf{pad} \leftarrow (\mathsf{PPPRF.Eval}(\mathsf{msk}_1, x) \| \cdots \| \mathsf{PPPRF.Eval}(\mathsf{msk}_{\ell_{\text{out}}}, x)) \oplus v$
Let $\mathsf{sk} \leftarrow (\mathsf{sk}_1, \cdots, \mathsf{sk}_{\ell_{\text{out}}}, \mathsf{pad})$
Output $\mathcal{A}(\mathsf{sk})$

Experiment Hyb_3. Recall that in Hyb_2, we use the unprogrammed keys to compute pad, that is, $\mathsf{pad} \leftarrow (\mathsf{PPPRF.Eval}(\mathsf{msk}_1, x) \| \cdots \| \mathsf{PPPRF.Eval}(\mathsf{msk}_{\ell_{\text{out}}}, x)) \oplus v$. In Hyb_3, we instead use the programmed keys to compute pad, that is, $\mathsf{pad} \leftarrow (\mathsf{PPPRF.PEval}(\mathsf{sk}_1, x) \| \cdots \| \mathsf{PPPRF.PEval}(\mathsf{sk}_m, x)) \oplus v'$.

Experiment Hyb_3

Let $x, f \leftarrow \mathcal{A}$
Let $v \xleftarrow{\$} \{0,1\}^{\ell_{\text{out}}}$
Let $v' \leftarrow f(v)$
Sample $\mathsf{msk}_1, \cdots, \mathsf{msk}_{\ell_{\text{out}}} \leftarrow \mathsf{PPPRF.Gen}(1^\lambda, \ell_{\text{in}}, 1)$
For each $1 \le i \le \ell_{\text{out}}$, let $\mathsf{sk}_i \leftarrow \mathsf{PPPRF.Prog}(\mathsf{msk}_i, x, \mathsf{PPPRF.Eval}(\mathsf{msk}_i, x) \oplus v_i \oplus v'_i)$
$\mathsf{pad} \leftarrow (\mathsf{PPPRF.PEval}(\mathsf{sk}_1, x) \| \cdots \| \mathsf{PPPRF.PEval}(\mathsf{sk}_m, x)) \oplus v'$
Let $\mathsf{sk} \leftarrow (\mathsf{sk}_1, \cdots, \mathsf{sk}_{\ell_{\text{out}}}, \mathsf{pad})$
Output $\mathcal{A}(\mathsf{sk})$

As long as programming correctness is satisfied, using either the unprogrammed or programmed keys to compute pad would yield the same result. Since programming correctness is satisfied with $1 - \mathsf{negl}(\lambda)$ probability, Hyb_2 and Hyb_3 have negligible statistical distance.

Experiment Hyb_4. In the earlier Hyb_3, the experiment programs the keys such that the programmed values are equal to the original evaluation outcome XOR'ed with $v \oplus v'$. In Hyb_4, instead of honestly computing the programmed keys, we generate the programmed keys using the simulator Sim.

Experiment Hyb_4

Let $x, f \leftarrow \mathcal{A}$
Let $v \xleftarrow{\$} \{0,1\}^{\ell_{\text{out}}}$
Let $v' \leftarrow f(v)$
Sample $\mathsf{sk}_1, \cdots, \mathsf{sk}_{\ell_{\text{out}}} \leftarrow \mathsf{Sim}(1^\lambda, \ell_{\text{in}}, 1)$
$\mathsf{pad} \leftarrow (\mathsf{PPPRF.PEval}(\mathsf{sk}_1, x) \| \cdots \| \mathsf{PPPRF.PEval}(\mathsf{sk}_{\ell_{\text{out}}}, x)) \oplus v'$
Let $\mathsf{sk} \leftarrow (\mathsf{sk}_1, \cdots, \mathsf{sk}_{\ell_{\text{out}}}, \mathsf{pad})$
Output $\mathcal{A}(\mathsf{sk})$

As long as f is PPT, and the underlying PPPRF satisfies δ-XOR functional programming privacy, Hyb_3 and Hyb_4 are $(\ell_{\text{out}} \cdot \delta)$-computationally indistinguishable. Specifically, we can go through ℓ_{out} inner-hybrids to replace the programmed keys one by one to a simulated key. In each hybrid step, we incur an δ loss in privacy.

Experiment Hyb_5. In Hyb_4, v is only used to compute v' and it is never used afterwards. In Hyb_5, we will directly sample v' at random without sampling v. Since the function $f(v)$'s output is δ'-indistinguishable from random upon a uniform input v, Hyb_4 and Hyb_5 are δ'-computationally indistinguishable.

Experiment Hyb_5

Let $x, f \leftarrow \mathcal{A}$
Let $v' \xleftarrow{\$} \{0,1\}^{\ell_{\text{out}}}$
Sample $\mathsf{sk}_1, \cdots, \mathsf{sk}_{\ell_{\text{out}}} \leftarrow \mathsf{PPPRF.Sim}(1^\lambda, \ell_{\text{in}}, 1)$
$\mathsf{pad} \leftarrow (\mathsf{PPPRF.PEval}(\mathsf{sk}_1, x) \| \cdots \| \mathsf{PPPRF.PEval}(\mathsf{sk}_{\ell_{\text{out}}}, x)) \oplus v'$
Let $\mathsf{sk} \leftarrow (\mathsf{sk}_1, \cdots, \mathsf{sk}_{\ell_{\text{out}}}, \mathsf{pad})$
Output $\mathcal{A}(\mathsf{sk})$

Finally, it is not hard to see that Hyb_5 is identically distributed as the ideal experiment $\mathsf{IdealProgPriv}$ where the simulator Sim outputs $\mathsf{sk}_1, \ldots, \mathsf{sk}_{\ell_{\text{out}}}$ sampled from the underlying PPPRF's simulator $\mathsf{PPPRF.Sim}(1^\lambda, \ell_{\text{in}}, 1)$ and a random $\mathsf{pad} \xleftarrow{\$} \{0,1\}^{\ell_{\text{out}}}$.

Summarizing the above, $\mathsf{RealProgPriv}$ is $(\delta' + \ell_{\text{out}} \cdot \delta)$-indistinguishable from $\mathsf{IdealProgPriv}$.

Proof of Theorem 5.4. Given the proof of Theorem 5.3 and the simulator construction, the proof of Theorem 5.4 is straightforward due to Theorem 5.2.

C Deferred Proofs for Section 6

C.1 Proof of Theorem 6.1

Henceforth, define the short-hand

$$\mathsf{PPPRF.FProg}(\mathsf{msk}, x, f) := \mathsf{PPPRF.Prog}\big(\mathsf{msk}, x, f(\mathsf{PPPRF.Eval}(\mathsf{msk}, x))\big).$$

Further, given some x, define the following functions:

- $f_D^{x,g}(v)$: output $g(v)$.
- $f_{D-i}^{x,g}(\mathsf{msk}_{D-i}, \mathsf{sk}_{D-i}, \tau_{D-i})$ where $i \in [D-1]$: output $(\mathcal{U}_\lambda, \mathsf{PPPRF}^{D-i}.\mathsf{FProg}(\mathsf{msk}_{D-i}, x_{D-i}, f_{D-i+1}), 1 - \tau_{D-i})$ where x_{D-i} is the $(D-i)$-th digit of the base-γ representation of x.

Using the above notation, the real experiment $\mathsf{Real}^{\mathcal{A}}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ in Theorem 6.1 can be equivalently rewritten as the following.

Experiment $\mathsf{Real}^{\mathcal{A}}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$. $\mathcal{A}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ outputs x, g. Sample $(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0) \leftarrow \mathsf{PPPRFTree.Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ subject to $T^x(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0) = 0^D$. Let $\mathsf{sk} \leftarrow \mathsf{PPPRF}^0.\mathsf{FProg}((\mathsf{msk}_0, \mathsf{sk}_0, \tau_0), f_1^{x,g})$, let $\mathsf{msk}' \leftarrow (\mathcal{U}_\lambda, \mathsf{sk}, 1 - \tau_0)$, and output $\mathcal{A}(\mathsf{msk}')$.

We want to show that $\mathsf{Real}^{\mathcal{A}}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ is $2^D \cdot \log^{1.1} \lambda \cdot (\delta' + D\delta)$-computationally indistinguishable from $\mathsf{Ideal}^{\mathcal{A}}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ as long as g is δ'-random-to-random. We will prove this through a sequence of hybrid experiments.

Experiment Hyb_0. Hyb_0 is otherwise identical as Real except that if the rejection sampling $(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0) \leftarrow \mathsf{PPPRFTree.Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ subject to $T^x(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0) = 0^D$ has not completed in $M = \log^{1.1} \lambda \cdot 2^D$ number of tries, simply abort outputting $\perp$.

Claim C.1. Suppose that $\mathsf{PPPRF}^1, \ldots, \mathsf{PPPRF}^{D-1}$ all satisfy pseudorandomness. Then, the distribution $T^x(\mathsf{msk})$ for a random string msk of appropriate length is computationally indistinguishable from a uniform distribution on $\{0,1\}^Z$. This implies that for any vector $\mathbf{b} \in \{0,1\}^Z$, the probability that a random msk of appropriate length satisfies $T^x(\mathsf{msk}) = \mathbf{b}$ is negligibly apart from $1/2^D$.

Proof. This can be proven through a straightforward hybrid argument, where we replace PPPRF^1 to PPPRF^{D-1} with a random oracle one by one when evaluating the function $T^x(\mathsf{msk})$. Due to the pseudorandomness of the underlying PPPRFs, the distributions of $T^x(\mathsf{msk})$ in adjacent pair of hybrids are computationally distinguishable. When we have replaced all of PPPRF^1 to PPPRF^{D-1} with random oracles, then the probability $T^x(\mathsf{msk}) = \mathbf{b}$ is exactly $1/2^D$ for every $\mathbf{b}$. □

Due to Claim C.1, the probability that Hyb_0 exhausts all $M = 2^D \log^{1.1} \lambda$ tries and outputs $\perp$ at the end is at most

$$\left(1 - 1/2^D + \mathsf{negl}(\lambda)\right)^{2^D \log^{1.1} \lambda} \leq (1 - 1/2^{D+1})^{2^{D+1} \cdot \frac{1}{2} \cdot \log^{1.1} \lambda} \leq 1/e^{\frac{1}{2} \cdot \log^{1.1} \lambda} = \mathsf{negl}(\lambda) \tag{1}$$

Thus, Hyb_0 and Real have only negligible statistical distance.

Experiment Hyb_1. Hyb_1 is just an equivalent rewrite of Real by deferring $T^x(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0) = 0^D$ check to the very end. Specifically, Hyb_1 is the following experiment: $\mathcal{A}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ outputs x, g. Repeat the following M times:

- sample $(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0) \leftarrow \mathsf{PPPRFTree.Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$;
- let $\mathsf{sk} \leftarrow \mathsf{PPPRF}^0.\mathsf{FProg}((\mathsf{msk}_0, \mathsf{sk}_0, \tau_0), f_1^{x,g})$, and let $\mathsf{msk}' \leftarrow (\mathcal{U}_\lambda, \mathsf{sk}, 1 - \tau_0)$,
- if $T^x(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0) = 0^D$, return and output $\mathcal{A}(\mathsf{msk}')$.

Output $\perp$.

Experiment Hyb_2. In Hyb_2, we change checking $T^x(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0) = 0^D$ to checking that $T^x(\mathsf{msk}') = 1^D$. Specifically, Hyb_2 is the following experiment: $\mathcal{A}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ outputs x, g. Repeat the following M times:

- sample $(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0) \leftarrow \mathsf{PPPRFTree.Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$;
- let $\mathsf{sk} \leftarrow \mathsf{PPPRF}^0.\mathsf{FProg}((\mathsf{msk}_0, \mathsf{sk}_0, \tau_0), f_1^{x,g})$, and let $\mathsf{msk}' \leftarrow (\mathcal{U}_\lambda, \mathsf{sk}, 1 - \tau_0)$,
- if $T^x(\mathsf{msk}') = 1^D$, return and output $\mathcal{A}(\mathsf{msk}')$.

Output $\perp$.

Assuming that the underlying PPPRF schemes all satisfy programming correctness, except with negligible probability in the random experiment Hyb_2, $T^x(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0) = 0^D \Leftrightarrow T^x(\mathsf{msk}') = 1^D$ holds for all up to M tries. Therefore, Hyb_1 and Hyb_2 have negligible statistical distance.

Experiment Hyb_3. In Hyb_3, we replace the sampling of the programmed key sk with a random string of appropriate length. Specifically, Hyb_3 is the following experiment: $\mathcal{A}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ outputs x, g. Repeat M times:

- sample $(\mathsf{msk}_0, \mathsf{sk}_0, \tau_0) \leftarrow \mathsf{PPPRFTree.Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$;
- let sk be a random string of appropriate length, and let $\mathsf{msk}' \leftarrow (\mathcal{U}_\lambda, \mathsf{sk}, 1 - \tau_0)$,
- if $T^x(\mathsf{msk}') = 1^D$, return and output $\mathcal{A}(\mathsf{msk}')$.

Output $\perp$.

Claim C.2. *Suppose that for every $i \in \{0, 1, \ldots, D-1\}$, PPPRF^i satisfies δ-general functional programming privacy. Then, Hyb_3 is $M \cdot (\delta' + D \cdot \delta)$-computationally indistinguishable from Hyb_2.*

Proof. By our assumption, $f_D^{x,g} = g$ is δ'-random-to-random. By the δ-general functional programming privacy of PPPRF^{D-1}, we have that $f_{D-1}^{x,g}$ is $(\delta' + \delta)$-random-to-random. Similarly, by the δ-general functional programming privacy of PPPRF^{D-2}, $f_{D-2}^{x,g}$ is $(\delta' + 2\delta)$-random-to-random. By induction, we have that $f_1^{x,g}$ is $(\delta' + (D-1)\delta)$-random-to-random. Finally, by the δ-general functional programming privacy of PPPRF^0, it holds that each sk in Hyb_1 is $(\delta' + D \cdot \delta)$-computationally indistinguishable from a random string of appopriate length. Because both experiments sample at most M such sk's, it holds that Hyb_3 is $M \cdot (\delta' + D \cdot \delta)$-computationally indistinguishable from Hyb_2. □

Experiment Hyb_4. Hyb_4 is the following experiment: $\mathcal{A}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ outputs x, g. Repeat M times:

- sample a random string msk' of appropriate length,
- if $T^x(\mathsf{msk}') = 1^D$, return and output $\mathcal{A}(\mathsf{msk}')$.

Output $\perp$.

It is not hard to see that Hyb_4 is an equivalent rewrite of Hyb_3, since msk_0 and τ_0 are sampled uniform at random in $\mathsf{PPPRF.Gen}$.

Observe that the only difference between Hyb_4 and Ideal is that Hyb_4 limits the number of tries to M and Ideal does not. Due to Claim C.1 and Equation (1), as long as $\mathsf{PPPRF}^1, \ldots, \mathsf{PPPRF}^{D-1}$ all satisfy pseudorandomness, the probability that Hyb_4 exhausts all M tries and outputs $\perp$ at the end is negligibly small. Therefore, Hyb_4 and Ideal have negligible statistical distance.

C.2 Proof of Theorem 6.2

Programming correctness is guaranteed if there exists a tree among the Z trees whose key is denoted msk_i, such that $T^x(\mathsf{msk}_i) = 1^D$. Due to Claim C.1, the probability that this happens is negligibly apart from $1/2^D$. Given the choice of Z, it is not hard to see that the probability that there exists such a tree is $1 - \mathsf{negl}(\lambda)$.

Henceforth, we focus on proving that our forest-based PPPRF satisfies functional programming privacy. Our proof goes through a sequence of hybrid experiments.

Experiment RealProgPriv. Recall that in $\mathsf{RealProgPriv}$, the adversary $\mathcal{A}$ submits $x, f_{\text{xor}}^{\Delta}(v) = v + \Delta$, and the challenger samples a random msk and programs it at x to be $f(v)$ where $v = \mathsf{Eval}(\mathsf{msk}, x)$. The resulting sk is returned to $\mathcal{A}$.

Experiment Hyb_1. Hyb_1 is otherwise identical to Real except that instead of sampling $\mathsf{msk}_i \leftarrow \mathsf{PPPRFTree.Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$, we sample $\mathsf{msk} := \{\mathsf{msk}_i\}_{i\in[Z]}$ as follows.

First, throw Z balls into 2^D bins at random. We number the bins using labels from $\{0,1\}^D$. If the bin 0^D is empty, return $\perp$ to $\mathcal{A}$. Else, continue with the following. For $j \in [Z]$, let $\boldsymbol{\beta}_j$ be the bin that ball j lands in, and sample $\mathsf{msk}_j \leftarrow \mathsf{PPPRFTree.Gen}(1^\lambda, \ell_{\text{in}}, \ell_{\text{out}})$ subject to $T^x(\mathsf{msk}_j) = \boldsymbol{\beta}_j$. Pick a random ball i that lands in bin 0^D. Let $v' = f(\mathsf{Eval}(\mathsf{msk}, x)) \oplus \mathsf{Eval}(\mathsf{msk}, x) \oplus \mathsf{PPPRFTree.Eval}(\mathsf{msk}_i, x)$, $\mathsf{sk}_i \leftarrow \mathsf{PPPRFTree.Prog}(\mathsf{msk}_i, x, v')$, and return the unordered set $\{\mathsf{msk}_1, \ldots, \mathsf{msk}_{i-1}, \mathsf{sk}_i, \mathsf{msk}_{i+1}, \ldots, \mathsf{msk}_Z\}$ to $\mathcal{A}$.

Claim C.3. Suppose that all the underlying $\mathsf{PPPRF}s$ in $\mathsf{PPPRFTrees}$ satisfy pseudorandomness. Then, Hyb_1 is computationally indistinguishable from Real.

Proof. Consider the following hybrid Hyb^* that is almost identical as Hyb_1 except that the balls into bins step is not using real randomness, but using the outcomes of Z randomly sampled $\mathsf{PPPRFTrees}$. Clearly, Hyb^* is identically distributed as Real. It suffices to show that Hyb^* is computationally indistinguishable from Hyb_1. By Claim C.1, the distribution of the bin loads are computationally indistinguishable whether we use true randomness or $\mathsf{PPPRFTrees}$. We can build an efficient reduction $\mathcal{R}$ that leverages an adverary $\mathcal{A}$ that can distinguish Hyb^* from Hyb_1 with non-negligible probability to distinguish the bin load distributions when sampled using true randomness vs. using $\mathsf{PPPRFTrees}$. Basically, the reduction obtains the bin loads, and then it continues to simulate Hyb^* for the adversary. If the rejection sampling exceeds some fixed polynomial number of tries, the reduction simply outputs 1. Otherwise, it gives the resulting $\{\mathsf{msk}_1, \ldots, \mathsf{msk}_{i-1}, \mathsf{sk}_i, \mathsf{msk}_{i+1}, \ldots, \mathsf{msk}_Z\}$ to $\mathcal{A}$ and outputs the same output as $\mathcal{A}$. The probability that the rejection sampling exceeds the time limit is negligibly small in Hyb_1; therefore, it should also be negligibly small in Hyb^* — otherwise we can easily build a reduction that leverages whether the rejection sampling completes in time to distinguish random bin loads from pseudorandom bin loads generated with $\mathsf{PPPRFTrees}$. It is not hard to show that if $\mathcal{A}$ has non-neligible advantage in distinguishing Hyb^* and Hyb_1, then $\mathcal{R}$ can distinguish random bin loads from pseudorandom bin loads generated with $\mathsf{PPPRFTrees}$ with non-negligible probability. □

Experiment Hyb_2. Hyb_2 is otherwise identical as Hyb_1 except that we now sample sk_i as a random string of appropriate length subject to $T^x(\mathsf{sk}_i) = 1^D$. Due to Theorem 6.1 and the fact that f^{Δ}_{xor} is random-to-random, Hyb_2 is $D \cdot 2^D \cdot \delta \cdot \log^{1.1} \lambda$-computationally indistinguishable from Hyb_1.

Hyb_2 can also be equivalently rewritten as the following. $\mathcal{A}$ outputs $x, f^{\Delta}_{\text{xor}}$. Using a random balls-into-bins experiment, sample $\boldsymbol{\beta}_j$ for all $j \in [Z]$. If the bin 0^D is empty, return $\perp$ to $\mathcal{A}$. Else, continue with the following. Choose a random i such that $\boldsymbol{\beta}_i = 0^D$, and reset $\boldsymbol{\beta}_i = 1^D$. For each $j \in [Z]$, sample a random string msk_j subject to $T^x(\mathsf{msk}_j) = \boldsymbol{\beta}_j$. Return $\{\mathsf{msk}_j\}_{j\in[Z]}$ to $\mathcal{A}$.

Experiment Hyb_3. Experiment Hyb_3 is almost identical to Hyb_2 except that we no longer replace $\boldsymbol{\beta}_i$ with 1^D.

Claim C.4. Hyb_3 *and* Hyb_2 *have statistical distance at most* $O(\sqrt{2^D/Z} \cdot \log^{1.2} \lambda)$.

Proof. It suffices to prove that the following distributions have at most $O(\sqrt{\nu/m} \cdot \log^{1.2} \lambda)$ statistical distance:

- Throw $m = \mathsf{poly}(\lambda)$ balls into $\nu \geq 2$ bins: if bin 0 is empty, output $\perp$; else output the vector of bin loads.

- Throw $m = \mathsf{poly}(\lambda)$ balls into $\nu \geq 2$ bins: if bin 0 is empty, output $\perp$; else move one ball from bin 0 to the last bin $\nu - 1$, and output the vector of bin loads.

  The probability of having bin loads $c_0, c_1, \ldots c_{\nu-1}$ after throwing m balls into ν bins is

$$\Pr[c_0, \ldots, c_{\nu-1}] = \frac{m!}{c_0! \ldots c_{\nu-1}! \nu^m}$$

Therefore,

$$\frac{\Pr[c_0 + 1, c_1, \ldots, c_{\nu-2}, c_{\nu-1} - 1]}{\Pr[c_0, c_1, \ldots, c_{\nu-2}, c_{\nu-1}]} = \frac{m!}{(c_0+1)! \ldots (c_{\nu-1}-1)! \nu^m} \cdot \frac{c_0! \ldots c_{\nu-1}! \nu^m}{m!} = \frac{c_{\nu-1}}{c_0 + 1}$$

Therefore, we have

$$\Pr[c_0 + 1, c_1, \ldots, c_{\nu-2}, c_{\nu-1} - 1] - \Pr[c_0, c_1, \ldots, c_{\nu-2}, c_{\nu-1}] = \Pr[c_0, \ldots, c_{\nu-1}] \cdot \left(\frac{c_{\nu-1}}{c_0 + 1} - 1 \right)$$

The statistical difference between the above two distributions is

$$\begin{aligned}
&\sum_{c_0, \ldots, c_{\nu-1}} |\Pr[c_0 + 1, c_1, \ldots, c_{\nu-2}, c_{\nu-1} - 1] - \Pr[c_0, c_1, \ldots, c_{\nu-2}, c_{\nu-1}]| \\
&= \sum_{c_0, \ldots, c_{\nu-1}} \Pr[c_0, \ldots, c_{\nu-1}] \cdot \left| \frac{c_{\nu-1}}{c_0 + 1} - 1 \right| \\
&= \sum_{c_0, c_{\nu-1}} \left| \frac{c_{\nu-1}}{c_0 + 1} - 1 \right| \cdot \sum_{c_1, \ldots, c_{\nu-2}} \Pr[c_0, \ldots, c_{\nu-1}] \\
&= \sum_{c_0, c_{\nu-1}} \left| \frac{c_{\nu-1}}{c_0 + 1} - 1 \right| \cdot \Pr[c_0, *, c_{\nu-1}] \qquad (\diamond)
\end{aligned}$$

By the Chernoff bound, except with negligible in λ probability, any bin's load lies within the range $R := [m/\nu - \sqrt{m/\nu} \log^{1.1} \lambda, m/\nu + \sqrt{m/\nu} \log^{1.1} \lambda]$. So we have

$$\begin{aligned}
(\diamond) &\leq \mathsf{negl}(\lambda) + \sum_{c_0, c_{\nu-1} \in R} \left| \frac{c_{\nu-1}}{c_0 + 1} - 1 \right| \cdot \Pr[c_0, *, c_{\nu-1}] \\
&\leq \mathsf{negl}(\lambda) + \sum_{c_0, c_{\nu-1} \in R} O\left(\frac{\sqrt{m/\nu} \log^{1.1} \lambda}{m/\nu} \right) \cdot \Pr[c_0, *, c_{\nu-1}] \\
&\leq \mathsf{negl}(\lambda) + O\left(\sqrt{\nu/m} \log^{1.1} \lambda \right) \cdot \sum_{c_0, c_{\nu-1} \in R} \Pr[c_0] \cdot \Pr[c_{\nu-1} | c_0] \\
&\leq \mathsf{negl}(\lambda) + O\left(\sqrt{\nu/m} \log^{1.1} \lambda \right) \leq O\left(\sqrt{\nu/m} \log^{1.2} \lambda \right)
\end{aligned}$$

□

Claim C.5. *Suppose all the underlying* PPPRF*s in* PPPRFTree *satisfy pseudorandomness.* Hyb_3 *is computationally indistinguishable from* IdealProgPriv *which simply samples Z keys of appropriate lengths and returns them to $\mathcal{A}$.*

Proof. The proof is almost the same as that of Claim C.3, and additionally using the fact that the probability that Hyb_3 aborts outputting $\perp$ is negligibly small. □

D Deferred Proofs for Section 7.1

D.1 Proof of Theorem 7.1

Proof. We now prove Theorem 7.1

Security. We first prove δ-security. Observe that the preprocessing just makes a streaming pass and leaks no information. Therefore we can focus on the query phase. Using the same proof as in Zhou et al. [ZPSZ24] and Ghoshal et al. [GZS24], we have the following claim:

Claim D.1. For every each time step t, at the end of the t-th query, the distribution of the PPPS keys in the client's hint table is the same as a collection of len freshly sampled PPPS keys, even when conditioned on the adversary $\mathcal{A}$'s view so far, and even when $\mathcal{A}$ can adaptively choose the queries.

Therefore, the adversary's view is identically distributed as if it is interacting with a client running the following stateless algorithm $\mathsf{QueryStateless}(1^\lambda, x, N)$: upon receiving query x, sample a PPPS key msk subject to $\mathsf{PPPS.Member}(\mathsf{msk}, x) = 1$, call $\mathsf{sk} \leftarrow \mathsf{PPPS.Program}(\mathsf{msk}, x, r)$ where $r \xleftarrow{\$} \{0, 1, \ldots, \sqrt{N} - 1\}$, and send sk to the server. To show δ-security of the PIR scheme, we construct the following simulator $\mathsf{Sim}(1^\lambda, N)$: sample a random PPPS key msk, and output msk. It suffices to show that for any x, the outputs of $\mathsf{QueryStateless}(1^\lambda, x, N)$ and $\mathsf{Sim}(1^\lambda, N)$ are δ-computationally indistinguishable. We can prove this through a sequence of hybrid experiments. Henceforth let $\mathsf{chunk}(x) = \lfloor x/\sqrt{N} \rfloor$.

$\mathsf{QueryStateless}(1^\lambda, x, N)$. Expanding $\mathsf{QueryStateless}(1^\lambda, x, N)$, it outputs the following distribution: sample s at random, sample a PPPRF key msk' subject to $\mathsf{PPPRF.Eval}(\mathsf{msk}', \mathsf{chunk}(x)) + s = x \mod \sqrt{N}$, let $\mathsf{sk}' \leftarrow \mathsf{PPPRF.Prog}(\mathsf{msk}', \mathsf{chunk}(x), r)$ for a random r, output (sk', s).

Experiment $\mathsf{Hyb}_1(1^\lambda, x, N)$. Sample a PPPRF key msk', sample s subject to $\mathsf{PPPRF.Eval}(\mathsf{msk}', \mathsf{chunk}(x)) + s = x \mod \sqrt{N}$, let $\mathsf{sk}' \leftarrow \mathsf{PPPRF.Prog}(\mathsf{msk}', \mathsf{chunk}(x), r)$ for a random r, output (sk', s).

Claim D.2. *If the* PPPRF *satisfies pseudorandomness, then* $\mathsf{Hyb}_1(1^\lambda, x, N)$ *is computationally indistinguishable from* $\mathsf{QueryStateless}(1^\lambda, x, N)$ *for any* x.

Proof. Hyb_1 is identically distributed as: sample msk at random, let $d = \mathsf{PPPRF.Eval}(\mathsf{msk}, \mathsf{chunk}(x))$, sample s such that $s + d = x \mod \sqrt{N}$, sample msk' subject to $\mathsf{PPPRF.Eval}(\mathsf{msk}', \mathsf{chunk}(x)) = d$, and output (msk', s). $\mathsf{QueryStateless}(1^\lambda, x, N)$ is identically distributed as the same experiment but d is sampled at random instead. Note that we can have both experiments abort if the rejection sampling does not complete within a fixed polynomial amount of time, and this only incurs negligible statistical distance. Now, the two experiments (with the abort modification) are computationally indistinguishable due to the pseudorandomness of the PPPRF. □

Experiment $\mathsf{Hyb}_2(1^\lambda, x, N)$. Sample a PPPRF key msk', sample s subject to $\mathsf{PPPRF.Eval}(\mathsf{msk}', \mathsf{chunk}(x)) + s = x \mod \sqrt{N}$, let

$$\mathsf{sk}' \leftarrow \mathsf{PPPRF.Prog}(\mathsf{msk}', \mathsf{chunk}(x), \mathsf{PPPRF.Eval}(\mathsf{msk}', \mathsf{chunk}(x)) + \Delta)$$

for a random Δ, output (sk', s). It is not hard to see that Hyb_2 is identically distributed as Hyb_1.

Experiment $\mathsf{Hyb}_3(1^\lambda, x, N)$**.** Sample a PPPRF key msk', let $\mathsf{sk}' \leftarrow \mathsf{PPPRF.Prog}(\mathsf{msk}', \mathsf{chunk}(x), \mathsf{PPPRF.Eval}(\mathsf{msk}', \mathsf{chunk}(x)) + \Delta)$ for a random Δ, sample s subject to $\mathsf{PPPRF.Eval}(\mathsf{sk}', \mathsf{chunk}(x)) + \Delta + s = x \mod \sqrt{N}$, output (sk', s). It is not hard to see that as long as the PPPRF satisfies programming correctness, then Hyb_3 and Hyb_2 has negligible statistical distance.

Experiment $\mathsf{Hyb}_4(1^\lambda, x, N)$**.** Sample a simulated key sk' using $\mathsf{PPPRF.Sim}$, sample s subject to $\mathsf{PPPRF.Eval}(\mathsf{sk}', \mathsf{chunk}(x)) + \Delta + s = x \mod \sqrt{N}$, output (sk', s). If PPPRF satisfies δ-XOR programming privacy, then Hyb_4 and Hyb_3 are δ-computationally indistinguishable through a straightforward reduction.

Finally, notice that Hyb_4 is identically distributed as the output of our simulator $\mathsf{Sim}(1^\lambda, N)$ described earlier since Δ is random.

Correctness. We next prove correctness. Correctness error can come from two bad events: 1) if no matched hint is found in step 2 or the matched hint is broken; and 2) if the replacement entries are depleted for the chunk of interest. It suffices to show that for each of the $\mathsf{poly}\log\lambda$ instances, for any fixed time step t, the probability that some bad event happens is constant.

We begin by bounding the first type of error. Fix an arbitrary time step t and an arbitrary query x_t that is distinct from all previous queries. Henceforth, let Γ_t denote the hint table right before time step t. Due to Claim D.1, all the queries made before the t-th time step must be independent of the PPPS keys in Γ_t. For each PPPS key in Γ_t, the probability that it fits a fixed query in the past is at most $1/\sqrt{N}$ where the probability comes from the distribution of the PPPS key. Therefore, the probability that it does not fit any query in the past is at most $p_1 = (1 - 1/\sqrt{N})^{\sqrt{N}} \approx 1/e$. If the PPPS key does not fit any past query, it cannot be broken. The probability that there does not exist a hint in Γ_t that contains x_t is $p_2 = (1 - 1/\sqrt{N})^{\mathsf{len}} \approx 1/e^C$. Therefore, the probability that for a single instance, the probability of no matched hint or matching a broken hint is at most $p_1 + p_2 \approx 1/e + 1/e^C$.

We next bound the second type of error. Here we need to use the fact that the database is permuted using a PRP upfront. Henceforth, we may assume that the PRP is replaced with a truly random permutation and this can cause only negligible difference in the error probability. Because of the random permutation, it suffices to bound the probability that among $\sqrt{N}$ random and distinct indices, strictly more than $\mathsf{poly}\log\lambda$ of them fall within the same chunk as x_t, This probability is negligibly small in λ due to standard tail bounds for the hypergeometric distribution.

□

D.2 Description of the two server PIR scheme

The 2-server scheme is a variant of the 1-server scheme. The only difference is that during preprocessing, instead of using a streaming pass to download the hint table and the replacement entries, the client contacts the *left* server to download the relevant information; and all the interactions during the query phase are with the *right* server. Specifically, during preprocessing, the client directly sends the $\{\mathsf{msk}_j\}_{j\in[\mathsf{len}]}$ keys to the left server, and for each key msk_j, the left server returns the parity of the database entries that belong to $\mathsf{PPPS.Set}(\mathsf{msk}_j, N)$. Further, the client sends the indices of the replacement entries to the left server, and get back the database entries at those indices.

Performance. We now analyze the performance of our 2-server preprocessing PIR. Again, we use $L_{\mathrm{PPPRF}} := L_{\mathrm{PPPRF}}(\lambda, \ell_{\mathrm{in}}, \ell_{\mathrm{out}}, \delta)$ denote an upper bound on the length of a normal and programmed

key and $T_{\text{PPPRF}} := T_{\text{PPPRF}}(\lambda, \ell_{\text{in}}, \ell_{\text{out}}, \delta)$ denote an upper bound on the evaluation and programming cost of the underlying PPPRF scheme.

- *Space.* Like the 1-server scheme, the client space is bounded by $\sqrt{N} \cdot \mathsf{poly}\log\lambda \cdot L_{\text{PPPRF}}$, and each server's space requirement is N.
- *Preprocessing cost.* The preprocessing communication and client computation is bounded by $\mathsf{poly}\log\lambda \cdot \sqrt{N} \cdot L_{\text{PPPRF}}$. The left server's preprocessing computation is bounded by $\mathsf{poly}\log\lambda \cdot N \cdot T_{\text{PPPRF}}$. The right server is not involved in the preprocessing and has no cost.
- *Query cost.* We analyze the query cost of the unbounded scheme where we perform next phase's preprocessing in the current phase of $\sqrt{N}$ queries. For each query, the client's online computation is bounded by $\mathsf{poly}\log\lambda \cdot \sqrt{N} \cdot T_{\text{PPPRF}}$ except with $\mathsf{negl}(\lambda)$ probability, and the online communication is $\mathsf{poly}\log\lambda \cdot L_{\text{PPPRF}}$. The client's offline computation and communication is bounded by $\mathsf{poly}\log\lambda \cdot L_{\text{PPPRF}}$. Both servers' computation are bounded by $\mathsf{poly}\log\lambda \cdot \sqrt{N} \cdot T_{\text{PPPRF}}$ per query.

For the special case when $\delta = 1/\mathsf{poly}\log\lambda$ which is what we need in the security amplification later, and using the PPPRF scheme of Section 6.2, we have the following performance:

- *Space:* client space $= \sqrt{N} \cdot (\epsilon\lambda \log N)^{O(1/\epsilon)}$, server space $= N$;
- *Preprocessing:* client computation/bandwidth $= \sqrt{N} \cdot (\epsilon\lambda \log N)^{O(1/\epsilon)}$, left server computation $= N^{1+\epsilon}(\epsilon\lambda \log N)^{O(1/\epsilon)} \cdot T_{\text{PRG}}(\lambda)$, right server computation $= 0$;
- *Per-query cost (both online and offline):* bandwidth $= (\epsilon\lambda \log N)^{O(1/\epsilon)}$, client/server computation $= N^{\frac{1}{2}+\epsilon}(\epsilon\lambda \log N)^{O(1/\epsilon)} \cdot T_{\text{PRG}}(\lambda)$.

E Proofs of Security Amplification

We now provide the detailed proof of our security amplification.

Remark E.1 (Comparison with the security amplification of [BGIK22])**.** *Boyle et al. [BGIK22] propose a similar security amplification for distributed programmable point functions but without the d-fold decoys. In the first version of their paper dated August 15, 2022, they only upgrade from δ-indistinguishability to $N\delta$-strong-indistinguishability (see their Lemma 3), which would result in PIR with unacceptable $\mathsf{poly}(N)$ cost. In the second version of their paper dated April 16, 2023, they got rid of this N-factor blowup, but a careful inspection of their proof suggests that the proof is incomplete. In particular, without the N-factor blowup, their proof only works if the hardcores for $\mathsf{PIR.Query}(0)$, $\mathsf{PIR.Query}(1)$, ..., $\mathsf{PIR.Query}(N-1)$ are the same, which is not true in general. With some non-trivial changes to their proof, it might be possible to fix this issue by directly proving that their underlying distributed point function satisfies strong δ-indistinguishability. In our paper, instead of directly proving that our underlying PPPRF satisfies strong δ-indistinguishability, we go through the d-fold decoy upgrade, giving a stronger and more general amplification theorem which may be of independent interest.*

E.1 Additional Preliminaries

Hardcore lemma for computational indistinguishability. Mauer and Tessaro [MT10] proved the following hardcore lemma for computational indistinguishability. Slightly informally, this hardcore theorem says for any two ensembles D_1 and D_2 that are δ-computationally indistinguishable,

there is a hardcore of size roughly $1-\delta$, such that no efficient adversary can effectively distinguish D_1 and D_2 sampled conditioned on being in this hardcore.

Theorem E.2 (Hardcore lemma for computational indistinguishability)**.** *Let $\ell_1(\cdot)$ and $\ell_2(\cdot)$ be polynomially bounded functions in λ. Let $D_1(1^\lambda,\cdot), D_2(1^\lambda,\cdot) : \{0,1\}^{\ell_1(\lambda)} \to \{0,1\}^{\ell_2(\lambda)}$ be two ensembles parametrized by λ. Suppose that there exists some $\delta(\lambda)$ such that for all non-uniform PPT adversaries $\mathcal{A}$, for sufficiently large λ,*

$$\left| \Pr_{r \xleftarrow{\$} \{0,1\}^{\ell_1(\lambda)}} [\mathcal{A}(1^\lambda, D_1(1^\lambda, r)) = 1] - \Pr_{r \xleftarrow{\$} \{0,1\}^{\ell_1(\lambda)}} [\mathcal{A}(1^\lambda, D_2(1^\lambda, r)) = 1] \right| \leq \delta(\lambda)$$

Then, there exists a hardcore $\mathsf{HC}(\lambda) \subseteq \{0,1\}^{\ell_1(\lambda)}$ whose size $|\mathsf{HC}(\lambda)| \geq (1-\delta(\lambda)) \cdot 2^{\ell_1(\lambda)}$, and for any non-uniform PPT adversary $\mathcal{A}'$, there exists a negligible function $\mathsf{negl}(\cdot)$, such that for every λ,

$$\left| \Pr_{r \xleftarrow{\$} \mathsf{HC}(\lambda)} [\mathcal{A}'(1^\lambda, D_1(1^\lambda, r)) = 1] - \Pr_{r \xleftarrow{\$} \mathsf{HC}(\lambda)} [\mathcal{A}'(1^\lambda, D_2(1^\lambda, r)) = 1] \right| \leq \mathsf{negl}(\lambda)$$

E.2 Strong Computational Indistinguishability and Technical Lemma

As a stepping stone in the proof of our security amplification theorem, we will work with a strengthened notion of δ-computational indistinguishability defined below.

Definition E.3 (Strong δ-computational indistinguishability)**.** We say that the ensemble $X(1^\lambda)$ is strongly δ-computationally indistinguishable from $Y(1^\lambda)$ if there exists $Z(1^\lambda)$ such that $X(1^\lambda)$ is computationally indistinguishable from $Z(1^\lambda)$, and moreover, $Z(1^\lambda)$ can be equivalently rewritten as: with probability $1-\delta$, output a random sample from $Y(1^\lambda)$; else output a random sample from any residual distribution.

Note that unlike δ-computational indistinguishability which is a symmetric notion, strong δ-computational indistinguishability is not symmetric.

The following technical lemma will be useful later in our security amplification construction and proof.

Lemma E.4 (Technical lemma)**.** *Let $\ell(\cdot)$ be a polynomially bounded function. Suppose $\{r \xleftarrow{\$} \{0,1\}^{\ell(\lambda)} : D(1^\lambda, r)\}$ and $\{r \xleftarrow{\$} \{0,1\}^{\ell(\lambda)} : D'(1^\lambda, r)\}$ are δ-computationally indistinguishable where $\delta \geq 1/\lambda^C$ for some constant C. Then, for $(1-1.1\delta)d > \log^2 \lambda$, the following ensemble $X'(1^\lambda)$ is strongly $\left(2\delta + \frac{\log \lambda}{\sqrt{d}}\right)$- computationally indistinguishable from $X(1^\lambda)$:*

- *$X(1^\lambda)$: sample $r_1, \ldots, r_d \xleftarrow{\$} \{0,1\}^{\ell(\lambda)}$ and output*
$$\mathsf{Set}(\underbrace{D(1^\lambda, r_1), D(1^\lambda, r_2), \ldots, D(1^\lambda, r_d)}_{d});$$
- *$X'(1^\lambda)$: sample $r_1, \ldots, r_d \xleftarrow{\$} \{0,1\}^{\ell(\lambda)}$, and output*
$$\mathsf{Set}(\underbrace{D(1^\lambda, r_1), \ldots, D(1^\lambda, r_{d-1})}_{d-1}, D'(1^\lambda, r_d)).$$

Proof. We prove the lemma through a sequence of hybrid experiments. Let HC denote the hardcore for distribution $D(1^\lambda, r)$ and $D(1^\lambda, r')$ according to Theorem E.2. Henceforth let $p = |\mathsf{HC}|/2^\ell$. First, the distribution $X'(1^\lambda)$ can be rewritten as the following experiment Hyb':

Experiment Hyb'. In Hyb', we sample a random copy r to embed the special instance D', and for each $i \in [d]$, we sample a random β_i to decide whether the i-th copy should be sampled subject to the hardcore or not:

1. For $i \in [d]$, sample $\beta_i \xleftarrow{\$} \mathsf{Bernoulli}(p)$.
2. For $i \in [d]$, if $\beta_i = 1$, sample $r_i \xleftarrow{\$} \mathsf{HC}$; else sample $r_i \xleftarrow{\$} \{0,1\}^\ell \backslash \mathsf{HC}$.
3. Output $\mathsf{Set}(D(1^\lambda, r_1), \ldots, D(1^\lambda, r_{i-1}), D'(1^\lambda, r_i), D(1^\lambda, r_{i+1}), \ldots, D(1^\lambda, r_d))$.

Experment Hyb_1. In Hyb_1 we revere the order of steps 1 and 2 in Hyb'. As a result, the experiment becomes equivalent to first sampling a binomial random variable h deciding how many copies will be sampled subject to the hardcore, and then sampling a bit β with probability h/d that decides whether the special D' copy should be sampled subject to the hardcore:

1. Sample $h \xleftarrow{\$} \mathsf{Binomial}(d, p)$, sample $\beta \xleftarrow{\$} \mathsf{Bernoulli}(h/d)$.
2. If $\beta = 1$, then sample h copies of D subject to HC, sample $d - h - 1$ copies of D subject to the complement of HC, and sample 1 copy of D' subject to HC. Output the unordered set of the sampled values.

   Else, if $\beta = 0$, sample h copies of D subject to HC, and sample $d - h - 1$ copies of D subject to the complement of HC, and sample 1 copy of D' subject to the complement of HC. Output the unordered set of the sampled values.

Experment Hyb_2. In Hyb_2, if $\beta = 1$, we replace the sampling from D' subject to HC with sampling from D subject to HC:

1. Sample $h \xleftarrow{\$} \mathsf{Binomial}(d, p)$, sample $\beta \xleftarrow{\$} \mathsf{Bernoulli}(h/d)$.
2. If $\beta = 1$, then sample h copies of D subject to HC, and sample $d - h$ copies of D subject to the complement of HC. Output the unordered set of the sampled values.

   Else, if $\beta = 0$, sample h copies of D subject to HC, and sample $d - h - 1$ copies of D subject to the complement of HC, and sample 1 copy of D' subject to the complement of HC. Output the unordered set of the sampled values.

Due to Theorem E.2, Hyb_2 is computationally indistinguishable from Hyb_1. Moreover, we know that $p = |\mathsf{HC}|/2^\ell \geq 1 - 1.1\delta$, and $\mathbf{E}[h] = p \cdot d \geq (1 - 1.1\delta)d$. Henceforth, let $\mu = p \cdot d$. By the Chernoff bound, except with $\mathsf{negl}(\lambda)$ probability, it must be that $h \geq \mu - \sqrt{\mu}\log\lambda$. This means that in Hyb_2,

$$\begin{aligned}
\Pr[\beta = 1] &\geq (1 - \mathsf{negl}(\lambda)) \cdot \frac{\mu - \sqrt{\mu}\log\lambda}{d} \\
&= (1 - \mathsf{negl}(\lambda)) \cdot \frac{\mu - \sqrt{\mu}\log\lambda}{\mu/p} \\
&\geq (1 - 2\delta)(1 - \frac{\log\lambda}{\sqrt{\mu}}) \\
&\geq (1 - 2\delta)(1 - \frac{\log\lambda}{\sqrt{(1 - 1.1\delta)d}}) \\
&\geq 1 - 2\delta - \frac{\log\lambda}{\sqrt{d}}
\end{aligned}$$

Experiment Hyb_3. Hyb_3 is the following experiment:

1. Sample $h \xleftarrow{\$} \mathsf{Binomial}(d, p)$, if $h < \mu - \sqrt{\mu}\log\lambda$, abort outputting $\perp$.
2. Sample $\beta \leftarrow \mathsf{Bernoulli}(1 - 2\delta - \frac{\log\lambda}{\sqrt{d}})$.
3. If $\beta = 1$, sample h copies of D subject to HC, and sample $d - h$ copies of D subject to the complement of HC. Output the unordered set of the sampled values.
4. Else if $\beta = 0$, sample the output from some suitable residual distribution.

We claim that for some suitable residual distribution, Hyb_3 and Hyb_2 have negligible statistical distance. To see this, first, consider an intermediate hybrid Hyb_2' which is almost identical to Hyb_2 except that we abort outputting $\perp$ if $h < \mu - \sqrt{\mu}\log\lambda$. Hyb_2' has negligible statistical distance as Hyb_2. It is easy to see that Hyb_2 and Hyb_2' are identically distributed if we select an appropriate residual distribution.

Note that in Hyb_3, we can equivalently switch the order of steps 1 and 2. Further, we can omit the check on h and aborting step which introduces only negligible statistical distance. Therefore, Hyb_3 has negligible statistical distance from the following experiment:

Experiment Hyb_4. Hyb_4 is the following experiment:

1. Sample $\beta \leftarrow \mathsf{Bernoulli}(1 - 2\delta - \frac{\log\lambda}{\sqrt{d}})$.
2. Sample $h \xleftarrow{\$} \mathsf{Binomial}(d, p)$.
3. If $\beta = 1$, sample h copies of D subject to HC, and sample $d - h$ copies of D subject to the complement of HC. Output the unordered set of the sampled values.
4. Else if $\beta = 0$, sample the output from some suitable residual distribution.

Hyb_4 can be equivalently rewritten as the following:

Experiment Hyb_5. Hyb_5 is the following experiment:

1. Sample $\beta \leftarrow \mathsf{Bernoulli}(p')$ for any $p' \leq 1 - 2\delta - \frac{\log\lambda}{\sqrt{d}}$.
2. If $\beta = 1$, output d independent samples from D.
3. Else if $\beta = 0$, sample the output from some suitable residual distribution.

□

E.3 Proof of Theorem 8.2

We can now prove Theorem 8.2 with our technical lemma. Correctness follows in a straightforward fashion from the correctness of the underlying LDC and PIR. We therefore focus on proving security.

We consider the following sequence of hybrid experiments, suppose that $\mathcal{A}$ controls the server $i \in [k]$.

Experiment $\mathsf{PIRReal}$. This is the real security experiment described in Section 7.1. Basically, the client and the servers run the honest preprocessing algorithm. Afterwards, in every time step, $\mathcal{A}$ chooses some index, the client then computes the LDC queries $x_1, \ldots, x_Q$, and for each $j \in [Q]$, it sends to $\mathcal{A}$ one copy of $\mathsf{PIR.Query}_i(x_j)$ and $d - 1$ copies of $\mathsf{PIR.Sim}_i(1^\lambda, N)$.

Experiment $\mathsf{HybStateless}$**.** The only change from $\mathsf{PIRReal}$ is the following: in each time step, for $j \in [Q]$, the client sends one copy of $\mathsf{PIR.QueryStateless}_i(x_j)$ and $d-1$ copies of $\mathsf{PIR.Sim}_i(1^\lambda, N)$. By the security of the underlying PIR, $\mathcal{A}$'s views in $\mathsf{PIRReal}$ and $\mathsf{HybStateless}$ are computationally indistinguishable.

Experiment Hyb**.** After the honest preprocessing, during each time step, for each $j \in [Q]$: the client samples $\beta_j \leftarrow \mathsf{Bernoulli}(1-\delta')$ for $\delta' = \sigma/2Q$. If $\beta_j = 1$, the client sends d copies of $\mathsf{PIR.Sim}_i(1^\lambda, N)$; else, sample from some residual distribution that depends on x_j and send the outcome.

We claim that for sufficiently large λ, there exists some residual distribution such that Hyb and $\mathsf{HybStateless}$ are computationally indistinguishable. By the security of the underlying PIR, we have $\mathsf{PIR.Sim}_i(1^\lambda, N)$ and $\mathsf{PIR.QueryStateless}(x_j)$ are δ-computationally indistinguishable. The claim now follows from the proof of Theorem E.4 and a simple hybrid argument to step through the Q copies, and by observing that when we choose $\delta < \sigma/6Q$, $d > \log^2 \lambda/\delta^2$, then $\delta' > 2\delta + \frac{\log \lambda}{\sqrt{d}}$ for sufficiently large λ.

Experiment Hyb'**.** After the honest preprocessing, during each time step, sample all of $\beta_1, \ldots, \beta_Q$. If more than σ among $\{\beta_1, \ldots, \beta_Q\}$ are 0, then abort outputting $\perp$. Else, for each $j \in [Q]$ such that $\beta_j = 1$, send d copies of $\mathsf{PIR.Sim}_i(1^\lambda, N)$; for each j where $\beta_j = 0$, sample from some residual distribution that depends on x_j.

We show that Hyb' has negligible statistical distance from Hyb by showing that the probability of aborting is negligibly small. In expectation, the number of copies among $\{\beta_1, \ldots, \beta_Q\}$ that are 0 is bounded by $Q \cdot \delta' = Q \cdot \frac{\sigma}{2Q} = \sigma/2$. By the Chernoff bound, the probability that the number of 0 copies exceeds σ is negligibly small in λ.

Finally, observe that in Hyb', since for at most σ among the Q copies, we send some distribution dependent on x_j, and all other copies do not leak the corresponding x_j. Due to the σ-smoothness of the underlying LDC, the client in Hyb' can be implemented without knowing the actual query. So Hyb' also naturally defines a simulator that can simulate the i-th server's view without knowing the query.