Cryptography Meets Algorithms (15893) Lecture Notes

Lecture 6: 2-Server PIR from Distributed Point Functions Batch PIR

Scribe: Trevor Leong

February 20, 2024

1 Distributed Point Function and 2-server PIR with Logarithmic Communication

In earlier lectures, we learned an information-theoretic 2-server PIR scheme by Dvir and Gopi [DG16], which achieves $n^{O(\sqrt{\log\log n/\log n})}$ communication per query. In this lecture, we will show how to get a 2-server PIR with logarithmic communication relying only on a pseudorandom generator (PRG). This scheme is also interesting from a practical perspective since in practice, we can use AES to realize the PRG, and modern CPUs have hardware acceleration for evaluating AES.

Recall that from our undergraduate cryptography course, we know that the existence of a PRG is equivalent to the existence of a one-way function (OWF) [HILL99]. Also, from an earlier lecture, we learned that any 1-server classical PIR scheme with non-trivial bandwidth implies Oblivious Transfer which cannot be constructed in a blackbox manner from OWF [IR89]. Therefore, the scheme we will talk about today is in the 2-server setting.

1.1 Preliminary: Pseudorandom Generator

We will rely on a pseudorandom generator (PRG), which takes in a short random seed and expands the seed to a longer pseudorandom string.

Definition 1 (PRG)**.** *Let $\ell(\cdot)$ be a polynomial and let $G : \{0,1\}^n \to \{0,1\}^{\ell(n)}$ be a deterministic polynomial-time algorithm. G is a* PRG *if it has the following properties:*

- ***Expansion:*** $\forall n, \ell(n) > n$.
- ***Pseudorandomness:*** *for any probabilistic polynomial-time distinguisher D, there exists a negligible function* $\mathsf{negl}(\cdot)$*, such that*

$$\left| \Pr_{r \xleftarrow{\$} \{0,1\}^{\ell(n)}}[D(r) = 1] - \Pr_{s \xleftarrow{\$} \{0,1\}^{n}}[D(G(s)) = 1] \right| \leq \mathsf{negl}(n)$$

1.2 Distributed Point Function

Definition 2 (Point function)**.** *A point function parametrized by some point $x \in \{0,1\}^\ell$ is a function that evaluates to 1 at x, and evaluates to 0 everywhere else. We will henceforth use the notation $P_x\{0,1\}^\ell \to \{0,1\}$ to denote a point function. By definition, $P_x(x) = 1$ and $P_x(x') = 0$ for $x' \neq x$.*

Boyle, Gilboa and Ishai [BGI16] introduced the concept of a distributed point function. A distributed point function is a functional secret-sharing of a point function. In this lecture, we will specifically focus on a *2-way* secret sharing of a point function. Essentially, given some point function P_{x^*}, one can "secretly share" the function to two keys k_L, k_R. Then, for party $t \in \{L, R\}$ which receives the key k_t, it can evaluate the function on any point x and get a share of the outcome denoted $\mathsf{Eval}(k_t, x)$. It is guaranteed that in every point x, $\mathsf{Eval}(k_L, x) \oplus \mathsf{Eval}(k_R, x) = P_{x^*}(x)$. In other words, it is possible to combine the two outcome-shares to reconstruct the evaluation of the point function at any point. Finally, security of the DPF requires that each party $t \in \{L, R\}$ does not learn the "special point" (i.e., x^*) given its individual key k_t.

Definition 3 (2-share DPF)**.** *A DPF is a pair of possibly randomized algorithms (*Gen,Eval*) with the following syntax:*

- $\mathsf{Gen}(1^\lambda, x^*)$*: Outputs a pair of keys* k_L, k_R*.*
- $\mathsf{Eval}(1^\lambda, k, x)$*: Outputs the evaluation outcome* $y \in \{0, 1\}$*.*

Correctness. *Correctness requires that for any* λ*, any* ℓ*, any* $x^*, x \in \{0,1\}^\ell$*,*

$$\Pr\left[k_L, k_R \leftarrow \mathsf{Gen}(1^\lambda, x^*) : \mathsf{Eval}(k_L, x) \oplus \mathsf{Eval}(k_R, x) = P_{x^*}(x)\right] = 1$$

Security. *Security requires that there exists a probabilistic polynomial-time simulator* Sim*, such that for any* $\ell = \ell(\lambda)$ *that is a polynomial function in* λ*, and any* $x^* \in \{0,1\}^{\ell(\lambda)}$*, the following experiments are computationally indistinguishable for both* $t = L$ *and* $t = R$*:*

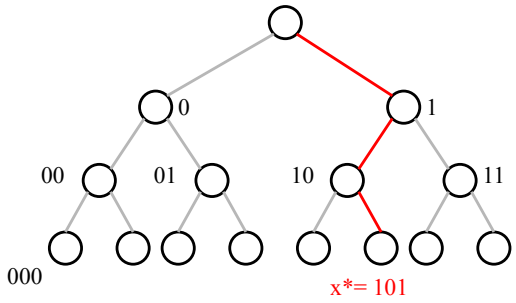
- $\mathsf{Real}(1^\lambda, x^*)$*:* $k_L, k_R \leftarrow \mathsf{Gen}(1^\lambda, x^*)$ *and output* k_t*;*
- $\mathsf{Ideal}(1^\lambda)$*: Output* $\mathsf{Sim}(1^\lambda, \ell)$*.*

Intuitively, security requires that the any individual key k_L or k_R can be simulated without knowledge of the special point x^*.

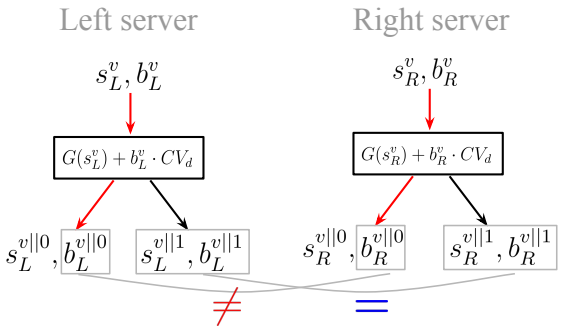
1.3 DPF $\Longrightarrow$ 2-Server PIR

Given a DPF scheme henceforth denoted $(\mathsf{DPF.Gen}, \mathsf{DPF.Eval})$, we can construct a 2-server PIR scheme as follows. Henceforth we use $\mathsf{DB} \in \{0,1\}^n$ to denote the database.

1. Given the query i, the client computes $(k_L, k_R) \leftarrow \mathsf{DPF.Gen}(1^\lambda, i)$
2. The client sends k_L to the left server and sends k_R to the right server.
3. Each server $t \in \{L, R\}$ receives k_t, and replies $y_t := \bigoplus_{j \in [n]} \mathsf{DPF.Eval}(k_t, j) \cdot \mathsf{DB}[j]$;
4. the client receives y_L and y_R from the two servers respectively, and outputs $y_L \oplus y_R$.



(a) The binary tree structure used in DPF. Each node has a unique name. The path from the root to the leaf node x^* is the "special path".



(b) Expansion during the evaluation algorithm. The key generation computes the correction vector CV in a way that guarantees the following: for any u not on the special path, $(s_L^u, b_L^u) = (s_R^u, b_R^u)$; and for any u on the special path, $b_L^u \neq b_R^u$ and moreover, the pair (s_L^u, s_R^u) is pseudorandom.

Figure 1: Demonstration of DPF construction.

Correctness. It is not hard to check that the answer output by the client is correct by the DPF's correctness:

$$\begin{aligned} y_0 \oplus y_1 &= \left(\bigoplus_j \mathsf{DB}[j] \cdot \mathsf{DPF.Eval}(k_0, j) \right) \oplus \left(\bigoplus_j \mathsf{DB}[j] \cdot \mathsf{DPF.Eval}(k_1, j) \right) \\ &= \left(\bigoplus_j \mathsf{DB}[j] \cdot (\mathsf{DPF.Eval}(k_0, j) \oplus \mathsf{DPF.Eval}(k_1, j)) \right) \\ &= \left(\bigoplus_j \mathsf{DB}[j] \cdot P_i(j) \right) \\ &= \mathsf{DB}[i]. \end{aligned}$$

Security. Security of the PIR follows directly from the security of the DPF.

1.4 DPF Construction

We now show how to construct an efficient DPF based on a pseudorandom generator G:

$$G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}.$$

The algorithm is based on a binary tree expansion idea.

Binary tree structure. Suppose we want to evaluate the DPF at n points denoted $0, 1, \ldots, n-1$, and we assume that n is a power of 2.

Imagine that there is a binary tree as depicted in Figure 1a. Each node in the tree has a name: the root's name is empty, and the two children of a node v are named $v||0$ and $v||1$, respectively. Henceforth, we say that the root is at *depth* 0, the leaves are at depth $\log n$, and so on. Each leaf node corresponds to a point in $\{0, 1, \ldots, n-1\}$. In particular, it helps to express each point in binary form. For a point function P_{x^*}, the path from the root to the leaf node x^* is called the special path highlighted in red in the figure.

Key structure and evaluation algorithm. The DPF's Gen algorithm outputs two keys $k_L = ((s_L, b_L), \mathsf{CV})$ and $k_R = ((s_R, b_R), \mathsf{CV})$, where

- s_L and s_R are both λ-bit PRG seeds;
- $b_L, b_R \in \{0,1\}$ are flags indicating whether correction is necessary during the key expansion (see use of the correction vector later in the algorithm). It is guaranteed that $b_L \neq b_R$; and
- $\mathsf{CV} = (\mathsf{CV}_1, \ldots, \mathsf{CV}_{\log n})$ is a correction vector.

We will focus on the left server's perspective for describing the evaluation algorithm. The right server's algorithm is the same except that its input is k_R instead of k_L. Imagine that initially, the root of the tree is associated with the pair (s_L, b_L) which comes from k_L. Starting from the root, we will perform a key expansion to compute a pair (s_L^v, b_L^v) for every node v in the tree. Each coordinate of the correction vector $\mathsf{CV}_1, \ldots, \mathsf{CV}_{\log n}$ will be consumed at each different level of the tree during the key expansion process.

More specifically, suppose some node v has the pair (s_L^v, b_L^v), we can compute the corresponding values at its two children $v||0$ and $v||1$ as follows where d denotes the depth of v's children:

$$(s_L^{v||0}, b_L^{v||0}), (s_L^{v||1}, b_L^{v||1}) \leftarrow G(s_L^v) \oplus \begin{cases} \mathbf{0} & \text{if } b_L^v = 0; \\ \mathsf{CV}_d & \text{if } b_L^v = 1. \end{cases}$$

Notice that the correction component CV_d is only applied if $b_L^v = 1$.

At the end of the expansion process, for each leaf node x, let b_L^x and b_R^x be the two bits associated with the leaf x output by the left and right servers, respectively. The outcome of the DPF at point x is then $b_L^x \oplus b_R^x$.

Key generation algorithm. The key generation algorithm samples random $s_L \xleftarrow{\$} \{0,1\}^\lambda$ and $s_R \xleftarrow{\$} \{0,1\}^\lambda$ at the root nodes for the left and right servers, respectively. It chooses a random bit $b_L \xleftarrow{\$} \{0,1\}$ and sets $b_R = b_L \oplus 1$. Then, it will choose the correction vector $\mathsf{CV}_1, \ldots, \mathsf{CV}_{\log n}$ in a way such that the following *invariants* are guaranteed for each tree node v:

- For every tree node v that is not on the special path leading to x^*, it must be that $(s_L^v, b_L^v) = (s_R^v, b_R^v)$.
- For every tree node v that is on the special path leading to x^*, it must be that $b_L^v = b_R^v \oplus 1$, and moreover, the joint distribution of the pair (s_L^v, s_R^v) is pseudorandom.

Note that for some node v, if $(s_L^v, b_L^v) = (s_R^v, b_R^v)$ is already guaranteed, then for any node u that is in the subtree of v, $(s_L^u, b_L^u) = (s_R^u, b_R^u)$ is automatically guaranteed because the left and right servers will behave identically when the apply the same expansion algorithm to compute all values in the subtree of v. Therefore, in the key generation algorithm, we can compute the correction vector CV using only the special path, to maintain the aforementioned invariants.

The detailed key generation algorithm is specified in Figure 2.

[elaine: TODO: add a remark about the following] Notice that if we only want to learn $\mathsf{Eval}(k, i)$ for a particular i, the computation cost will be $O(\log n)$ calls to the PRG, because we can focus on the path from the root to the i-th leaf and ignore other nodes. However, if we want to evaluate $\mathsf{Eval}(k, i)$ for all $i \in [n]$, the computation cost will be $O(n)$ because we can simply do the evaluation on the whole tree.

Generation Algorithm: $\mathsf{Gen}(1^\lambda, x^*)$

Initialization:

- Sample s_L, s_R as two λ-bit random strings.
- Sample a random bit b_L. Let $b_R = b_L \oplus 1$.
- Let $\{x^*[1], \ldots, x^*[\log n]\}$ be x^*'s binary representation.

Constructing correction vectors:

Initialize v to be an empty string.

For $i \in \{1, \ldots, \log n\}$:

- Sample a random string $r \xleftarrow{\$} \{0,1\}^\lambda$.
- Set $\mathsf{CV}_i \in \{0,1\}^{2\lambda+2}$ as an unknown variable and build the equation as follows.
  - Let $s_L^{v||0}||b_L^{v||0}||s_L^{v||1}||b_L^{v||1} \leftarrow G(s_L^v) \oplus (b_L^v \cdot \mathsf{CV}_i)$; *Expansion on the LHS*
  - Let $s_R^{v||0}||b_R^{v||0}||s_R^{v||1}||b_R^{v||1} \leftarrow G(s_R^v) \oplus (b_R^v \cdot \mathsf{CV}_i)$; *Expansion on the RHS*
  - If $x^*[i] = 0$: set the invariant as

$$(s_L^{v||0}||b_L^{v||0}||s_L^{v||1}||b_L^{v||1}) \oplus (s_R^{v||0}||b_R^{v||0}||s_R^{v||1}||b_R^{v||1}) = (r||1||0^\lambda||0).$$

  - If $x^*[i] = 1$: set the invariant as

$$(s_L^{v||0}||b_L^{v||0}||s_L^{v||1}||b_L^{v||1}) \oplus (s_R^{v||0}||b_R^{v||0}||s_R^{v||1}||b_R^{v||1}) = (0^\lambda||0||r||1).$$

  (Force the $x^*[i]$ direction to be different and the other direction to be identical.)
- Solve the equation based on the invariant to get CV_i. The equation is always solvable.
- Let $v \leftarrow v||x^*[i]$.

Output: Output k_L, k_R that include the strings and bits on the root and the shared correction vectors:

$$k_L = (s_L, b_L, \mathsf{CV}_1, \ldots, \mathsf{CV}_{\log n})$$
$$k_R = (s_R, b_R, \mathsf{CV}_1, \ldots, \mathsf{CV}_{\log n})$$

Figure 2: The 2-way binary DPF generation algorithm.

Generation Algorithm. The generation algorithm needs to generate two correlated keys, such that when we look at the bit at the x-th leaves on the left server and the right server, (say b_L^x and b_R^x), it must that $b_L^x \oplus b_R^x = P_{x^*}(x)$. In fact, we want to ensure the following stronger properties. Say on a particular node v, we denote the information evaluated by the left server and the right server as (s_L^v, b_L^v) and (s_R^v, b_R^v), respectively. We also denote the path from the root to the x^*-th leaf as the "special path".

1. If the node v is on the special path (i.e., v is a prefix of the binary representation of x^*), then $b_L^v \neq b_R^v$, and s_L^v and s_R^v are independent.
2. Otherwise, $(s_L^v, b_L^v) = (s_R^v, b_R^v)$.

The properties also hold for the leaf nodes, which is sufficient to prove the correctness of the DPF. We now see the how to ensure these properties. We have the first lemma, which can be proved by a simple induction argument.

Lemma 4. *If on some particular node v, $(s_L^v, b_L^v) = (s_R^v, b_R^v)$, then the information on the subtrees of v will be identical, regardless of $\mathsf{CV}_1, \ldots, \mathsf{CV}_{\log n}$. That is, for every v' that has prefix v, $(s_L^{v'}, b_L^{v'}) = (s_R^{v'}, b_R^{v'})$.*

This lemma shows that we only need to focus on the "special path". We can sample the roots first by sampling two random strings s_L, s_R and let b_L and b_R be two different bits. Then, we can just "simulate" the expansion process on the special path, and then set up the correction vector according to the target properties. Given a node v on the special path (v is a prefix of x^*), say v is on the $i-1$ level. Now assuming the correction vector CV_i is a *variable* to be determined. We can still following the evaluation algorithm, but all the expansion results, $s_L^{v||0}, b_L^{v||0}, s_L^{v||1}, b_L^{v||1}$, $s_R^{v||0}, b_R^{v||0}, s_R^{v||1}, b_R^{v||1}$, will be temporally some linear expressions of the variable CV_i. Then, we can enforce the invariants. Say $v||0$ is still the prefix of x (the other case is symmetric). We require that

- *Invariant on the special path.* 1) The xor-sum of $s_L^{v||0}$ and $s_R^{v||0}$ is a freshly sample random string r (ensuring they are independent); 2) $b_L^{v||0}$ and $b_L^{v||0}$ are different.
- *Invariant on other nodes.* $(s_L^{v||1}, b_L^{v||1}) = (s_R^{v||1}, b_R^{v||1})$.

Now these invariants set up a simple linear equation and we can easily solve the value of CV_i. The equation is always solvable because b_L^v and b_R^v are different. Thus, the CV_i will always and only affect the expansion on the one side. Note that the second invariant just enforces that the whole subtree of $v||1$ will be identical, and we just need to go to the child $v||0$ and derive CV_{i+1} based on $s_L^{v||0}, b_L^{v||0}$ and $s_R^{v||0}, b_R^{v||0}$. We run this algorithm from level 1 to level $\log n$ and get $\mathsf{CV}_1, \ldots, \mathsf{CV}_{\log n}$. The algorithm is presented in Figure 2 and an example is presented in Figure 1.

Analysis. The correctness can be verified by doing an induction proof from level 1 to level $\log n$. The security analysis is referred to [BGI16]. It is also clear to see that the key size is $\lambda + 1 + \log n \cdot (2\lambda + 2) = O_\lambda(\log n)$.

2 Batch-PIR

2.1 Motivation

So far, every PIR scheme we have seen only retrieves 1 bit at a time. In many use cases, the client may want to fetch up to Q database entries at the same time. The navie solution is

just to repeat the single-query algorithm Q times. For instance, to compute Q queries using a single-query PIR scheme with $O(n)$ computation, it requires $O(Qn)$ computation. Batch-PIR aims to handle multiple parallel queries at the same time, reducing the amortized cost.

2.2 A simple load-balancing scheme

Given an n bit long database and $Q = o(n)$ queries, we load-balance the database into $\frac{Q}{\log Q}$ buckets – we use a hash function to hash each database entry to a bucket (by hashing their index). Then, given the Q queries, we again use the hash function to place the queries to the buckets. In expectation, each bucket will have $\log Q$ queries. Based on the balls-into-bins argument, each bucket will have no more than $\lambda \log Q$ queries with $1 - \mathsf{negl}(\lambda)$ probability. Therefore, we just do $\lambda \log Q$ PIR queries in each bucket (possibly dummy queries) to retrieve the target entries. This ensures the success probability is at least $1 - \mathsf{negl}(\lambda)$.

Security. We are always making fix number of queries ($\lambda \log Q$ PIR queries in each bucket), so the scheme is secure by a reduction to the security of the underlying PIR scheme.

Cost. Moreover, say the underlying single-query PIR computation cost is linear in the size of the database. We are making $\lambda \log Q$ queries to each bucket, and the total size of the buckets is just n. Therefore, the total computation cost is $O(\lambda \log Qn)$. So if $Q > \lambda \log Q$, this simple scheme saves computation.

2.3 Cuckoo Hashing based scheme [ACLS18]

Angel et al. [ACLS18] proposed SealPIR that uses cuckoo hashing to do batch PIR.

Cuckoo Hashing

Definition 5 (Cuckoo hashing)**.** *Given n balls, b buckets, and w independent hash functions $h_0, \cdots, h_w$ that map a ball to a random bucket, compute w candidate buckets for each ball by applying the w hash functions. For each ball x, place x in any empty candidate bucket. If none of the w candidate buckets are empty, select one at random, remove the ball currently in that bucket (x_{old}), place x in the bucket, and re-insert x_{old}. If re-inserting x_{old} causes another ball to be removed, this process continues recursively until we finish the insertion or a maximum number of iterations is achieved.*

Batch PIR based on Cuckoo Hashing. The scheme is as follows.

- **Serer encoding.** Given an n bit database, b buckets, and w hash functions, we hash each entry in the database (using their index as the key) to all w candidate buckets and store it there. This results in a encoded database that each original entry is replicated w times. The server will share the hash functions to the client.
- **Client scheduling.** Given the Q queries, the client use the cuckoo hashing method to insert (or say, schedule) the queries to the buckets (again, using the indices as the keys). Our target is that each bucket has at most one query, and all query can be inserted in one of its candidate bucket.
- **Client query.**. Now the client just makes one query in each bucket. If the client successfully insert all queries earlier, it can then proceed to learn all the target entries because the server has inserted the entries in all the candidate buckets.

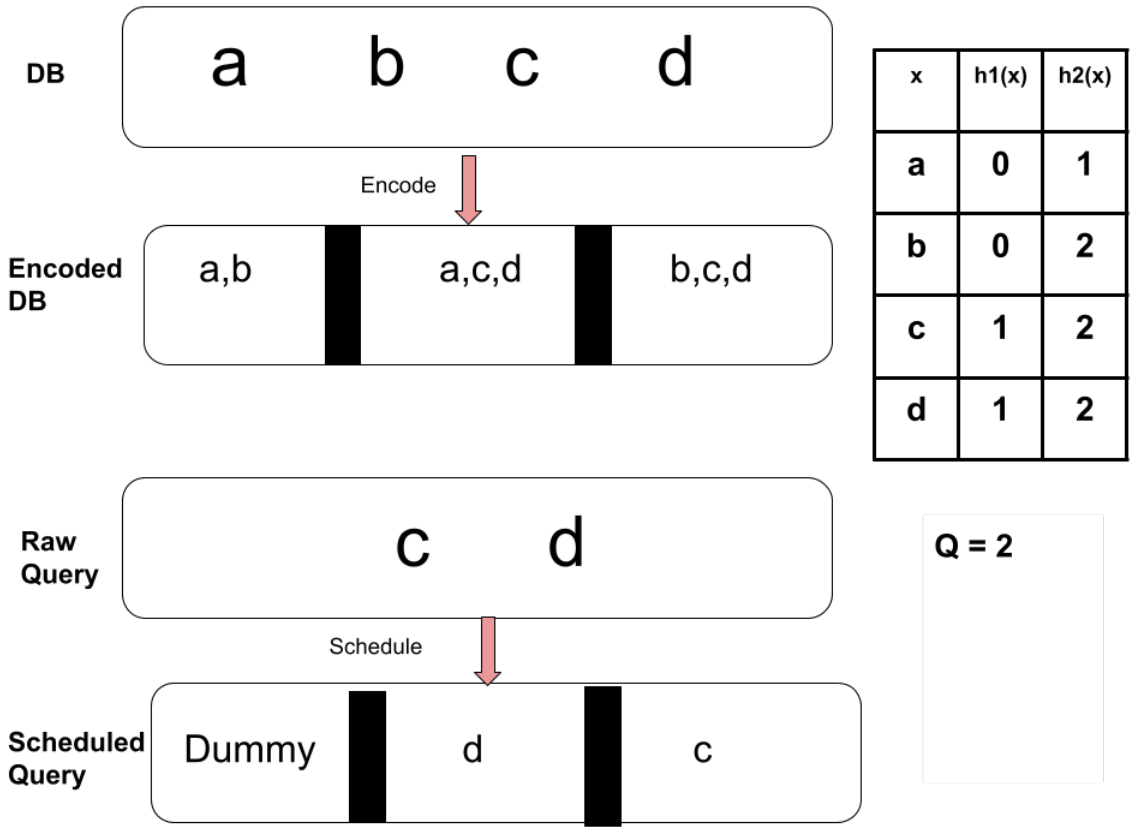


Figure 3: An example of the cuckoo hashing based batch PIR.

An example can be found in Figure 3.

The authors used 3 hash functions for encoding and set the number of buckets $b = 1.5Q$. For $Q \geq 200$, the author showed that the chance of failure during the scheduling phase is $\leq 2^{-40}$. Notice that this is not cryptographically neligible. To enforce negligible failure probability, we can introduce a size λ stash of the cuckoo hash table. That is, the stash stores at most λ elements that fail to be inserted. Then, the client also has to make additional λ PIR queries to the whole database. This ensures the failure probability to be $\mathsf{negl}(\lambda)$.

Cost Analysis. Assume the underlying single-query PIR scheme is linear. The client will make one query in each bucket and the total bucket size is wn. Also, the client needs to make λ additional query to the whole database to ensure negligible failure probability. Then, the computation cost for the Q queries are just $(w + \lambda)n$.

References

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. In *2018 IEEE symposium on security and privacy (SP)*, pages 962–979. IEEE, 2018.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1292–1303, 2016.

[DG16] Zeev Dvir and Sivakanth Gopi. 2-server pir with subpolynomial communication. *Journal of the ACM (JACM)*, 63(4):1–15, 2016.

[HILL99] Johan Håstad, Russell Impagliazzo, Leonid A Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.

[IR89] R. Impagliazzo and S. Rudich. Limits on the provable consequences of one-way permutations. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, STOC '89, page 44–61, New York, NY, USA, 1989. Association for Computing Machinery.