Cryptography meets algorithms (15893) Lecture Notes

Lecture 6: 2-server PIR with $O_\lambda(\log n)$ communication and Batch-PIR

Scribe: Trevor Leong

February 14, 2024

1 Distributed Point Function and 2-server PIR with $O_\lambda(\log(n))$ Communication [?]

We saw that in the IT-setting, the best known upper bound on the communication of 2-server PIR scheme is by Dvir and Gopi [?]. The scheme achieves $n^{O(\sqrt{\log\log n/\log n})}$ communication. In this lecture, we see that if we are willing to assume that one-way functions (OWF) exist, we can construct a 2-server PIR scheme with $O_\lambda(\log n)$ communication.

1.1 Basic Idea

This scheme use pseudorandom generators (PRG). In particular, PRG implies the existence of OWF and OWF implies the existence of PRG [?].

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

- ***Expansion:*** $\forall n, l(n) > n$.
- ***Pseudorandomness:*** *$\forall$ PPT distinguishers D, $\exists$ negl() s.t.*

$$|\Pr[D(r) = 1] - \Pr[D(G(s)) = 1]| \leq negl(n)$$

Where $r \xleftarrow{\$} \mathcal{U}^{l(n)}$ and $s \xleftarrow{\$} \mathcal{U}^n$.

Notation 1. *For $x, y \in \{0,1\}^*$, the point function $P_{x,y} : \{0,1\}^{|x|} \to \{0,1\}^{|x|}$ is defined by $P_{x,y}(x) = y$ and $P_{x,y}(x') = 0^{|y|}$ for all $x' \neq x$*

Boyle, Gilboa and Ishai [?] introduced the concept of distributed point function. We are going to focus on the case of a 2-share binary distributed point function. Essentially, given the full description of the point function $f = P_{x^*}$ (where $f(x^*) = 1$ and $f(x) = 0$ for all $x \neq x^*$), one can "secretly share" the function to two keys k_0, k_1. Then, if party $t \in \{0,1\}$ receives k_t, it can evaluate the key on all possible location x and get an evaluation of $\mathsf{Eval}(k_t, x)$. It is guaranteed that in every location x', $\mathsf{Eval}(k_0, x) \oplus \mathsf{Eval}(k_1, x) = f(x)$. Also, each party t has no information about the "special location" (i.e., x^*) given the key k_t.

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

- $\mathsf{Gen}(1^\lambda, x^*)$*: Outputs a pair of keys k_0, k_1.*
- $\mathsf{Eval}(1^\lambda, k, x)$*: Outputs $y \in \{0,1\}^*$*

Correctness. $\Pr[k_0, k_1 \leftarrow \mathsf{Gen}(1^\lambda, x^*) : \mathsf{Eval}(k_0, x) \oplus \mathsf{Eval}(k_1, x) = P_{x^*}(x)] = 1.$

Privacy. *Given $t \in \{0,1\}$ and any $x^* \in \{0,1\}^\ell$, there exists a PPT simulator Sim, such that the following experiments are computaionally indistinguishable:*

- $\mathsf{Real}(1^\lambda)$*: $k_0, k_1 \leftarrow \mathsf{Gen}(1^\lambda, x^*)$ and output k_t;*
- $\mathsf{Ideal}(1^\lambda)$*: Output $\mathsf{Sim}(1^\lambda, \ell)$.*

1.2 2-server PIR scheme based on DPF

Let $\mathsf{DB} \in \{0,1\}^n, i \in [n]$ and λ be the security parameter. Suppose we have an efficient 2-share binary DPF scheme $\mathsf{DPF.Gen}$ and $\mathsf{DPF.Eval}$, such that the key length is $O_\lambda(\log n)$ (where n is the size of the input domain), and the evaluation algorithm is efficient. Then, we can have the following 2-server PIR scheme.

1. Given the query i, Client computes $(k_0, k_1) \leftarrow \mathsf{DPF.Gen}(1^\lambda, i)$
2. Client sends (k_0, k_1) to Server 0 and Server 1 respectively.
3. For $t \in \{0,1\}$, Server t responds with $y_i \leftarrow \bigoplus_{j\in[n]} \mathsf{DPF.Eval}(k_t, j) \cdot \mathsf{DB}[j]$;
4. Client computes the answer as $y_0 \oplus y_1$.

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 follows from the security of the DPF.

1.2.1 DPF construction using PRG

Now we see how to construct this efficient 2-share binary DPF, based on a PRG G:

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

The algorithm is based on a binary tree expansion.

The key structure. The Gen algorithm outputs two keys, k_0, k_1. For each key key_t where $t \in \{0,1\}$, it contains the "root" information as a λ-bit random string s and a bit $b \in \{0,1\}$. It also contains $\log n$ "correction vectors" $\mathsf{CV}_1, \ldots, \mathsf{CV}_{\log n}$, each of size $2\lambda + 2$ bit. k_0, k_1 will share the same sets of correction vectors. We will see how they are constructed later.

Initialization:

- Sample s_0, s_1 as two random λ-bit random strings.
- Sample a random bit b_0. Let $b_1 = b_0 \oplus 1$.
- Let $k_0 = (s_0, b_0)$ and $k_1 = (s_1, b_1)$;
- Let $\{x^*[1], \ldots, x^*[\log n]\}$ be x^*'s binary representation.

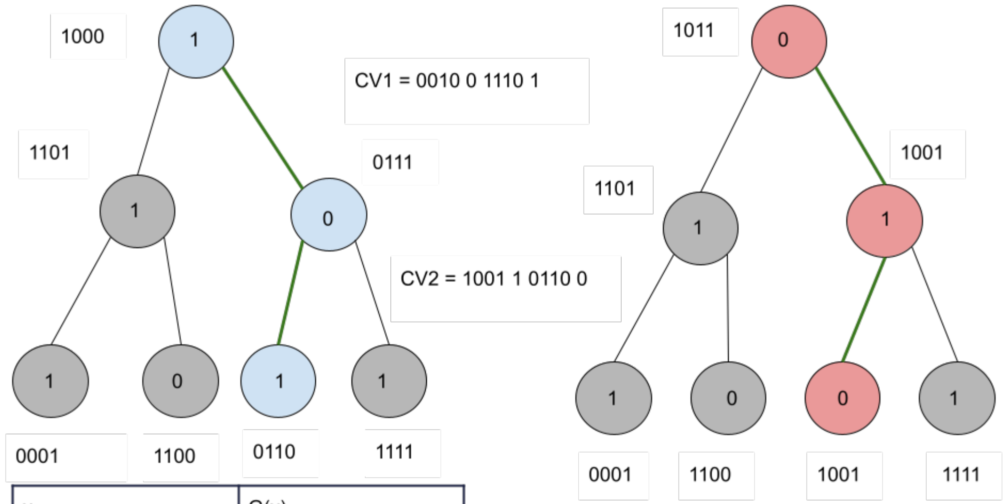
Constructing correction vectors: For $i \in \{1, \ldots, \log n\}$:

- Let $s_L^0||b_L^0||s_R^0||b_R^0 \leftarrow G(s_0)$;
- Let $s_L^1||b_L^1||s_R^1||b_R^1 \leftarrow G(s_1)$;
- Let $s_L^*||b_L^*||s_R^*||b_R^* = (s_L^0||b_L^0||s_R^0||b_R^0) \oplus (s_L^1||b_L^1||s_R^1||b_R^1)$;
- Sample r as a random λ-bit random string;
- If $x^*[i] = 0$: let $\mathsf{CV}_i = \ r \ ||b_L^* \oplus 1||s_R^*||b_R^*$; *If the path goes left;*
- If $x^*[i] = 1$: let $\mathsf{CV}_i = s_L^*||b_L^*|| \ r \ ||b_R^* \oplus 1$; *If the path goes right*;
- Expand $(s_0, b_0), (s_1, b_1)$ to their children as the same as the expansion algorithm, using CV_i. Update $(s_0, b_0), (s_1, b_1)$ by replacing them with their children's pairs (only the pairs on the special path).

Output :

- Attach $\mathsf{CV}_1, \ldots, \mathsf{CV}_{\log n}$ to k_0 and k_1.
- Output k_0, k_1.

Figure 1: The 2-share binary DPF generation algorithm.



x	G(x)
1000	1111 1 1001 1
1101	1000 1 1010 0
0111	0110 1 1111 1
1001	0000 1 1001 1
1011	1101 1 1001 1

$\lambda = 4$

Lshare = 1011, L root=(1000, 1)

RShare = 1001,R root = (1011, 0)

Figure 2: An example of the DPF construction.

Evaluation algorithm. Given a key containing s, b and $\mathsf{CV}_1, \ldots, \mathsf{CV}_{\log n}$, the evaluation algorithm is as follows. We consider a binary tree with $\log n + 1$ levels and n leaves (assume n is a power of 2). The root is on the 0-th level and the leaves are on the $\log n$-th level. Each node contains a key $s \in \{0,1\}^\lambda$ (a λ-bit random string) and a bit $b \in \{0,1\}$. Suppose now we want to expand the information of this node (on level $i-1$) to its two children (on levle i), we only need the following one-line expansion algorithm:

$$s_L||b_L||s_R||b_R \leftarrow G(s) \oplus \begin{cases} 0 & b = 0; \\ \mathsf{CV}_i & b = 1. \end{cases}$$

That is, s_L, b_L are the information on its left child, and s_R, b_R are the information on its right child. Finally, the bit on the i-th leaf will be $\mathsf{Eval}(\mathsf{key}, i)$.

Notice that if we only want to learn $\mathsf{Eval}(\mathsf{key}, i)$ 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 i-th leaf to the root and ignore other nodes. However, if we want to learn $\mathsf{Eval}(k, i)$ for all $i \in [n]$, the computation cost will be $O(n)$ because we can simply do the expansion on the whole tree.

Generation Algorithm. The generation algorithm needs to generate two correlated keys, such that when we look at the bit at the x-th leaves (say b_x^0 and b_x^1) on the trees expnaded by k_0 and k_1, it must that $b_x^0 \oplus b_x^1 = P_{x^*}(x)$. In fact, we want to ensure the following stronger properties. Say on a particular node, we denote the information expanded by k_0 and k_1 as (s_0, b_0) and (s_1, b_1), respectively.

1. If the node is on the "special path" (i.e., the path from the root to the x^*-th leaf), then $b_0 \neq b_1$, and s_0 and s_1 are independent.

2. Otherwise, $b_0 = b_1$ and $s_0 = s_1$.

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

Lemma 3. *If on some particular node, $b_0 = b_1$ and $s_0 = s_1$, then the subtrees expanded based on (s_0, b_0) and (s_1, b_1) will be identical, regardless of $CV_1, \ldots, CV_{\log n}$.*

This lemma shows that we only need to focus on the "special path". We can sample the root information first by sampling two random strings s_0, s_1 and let b_0 and b_1 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. Since on the i-th level of the special path, there will be only one side affected by the correction vector (because the bits are different). Therefore, we can always set the correction vector to ensure that after applying the correction vector, the children on the special path still have different bits and independent random strings, while the children deviating from the path will become identical. The algorithm is presented in Figure **??** and an example is presented in Figure **??**.

Analysis. The correctness can be verified by doing an induction proof from level 1 to level $\log n$. The privacy analysis is referred to [?]. 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. This works great for yes/no questions, but is cumbersome if we wanted to do anything practical. For instance, to compute Q queries using a naive PIR scheme with $O(n)$ computation, it requires $O(Qn)$ computation. Batch-PIR aims to make PIR more practical by enabling multiple responses for a single query, grouping together multiple responses into one and reduce 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 and place it there. 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 [?]

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

Cuckoo Hashing

Definition 4 (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.

An example can be found in Figure **??**.

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 $(w + \lambda)n$.

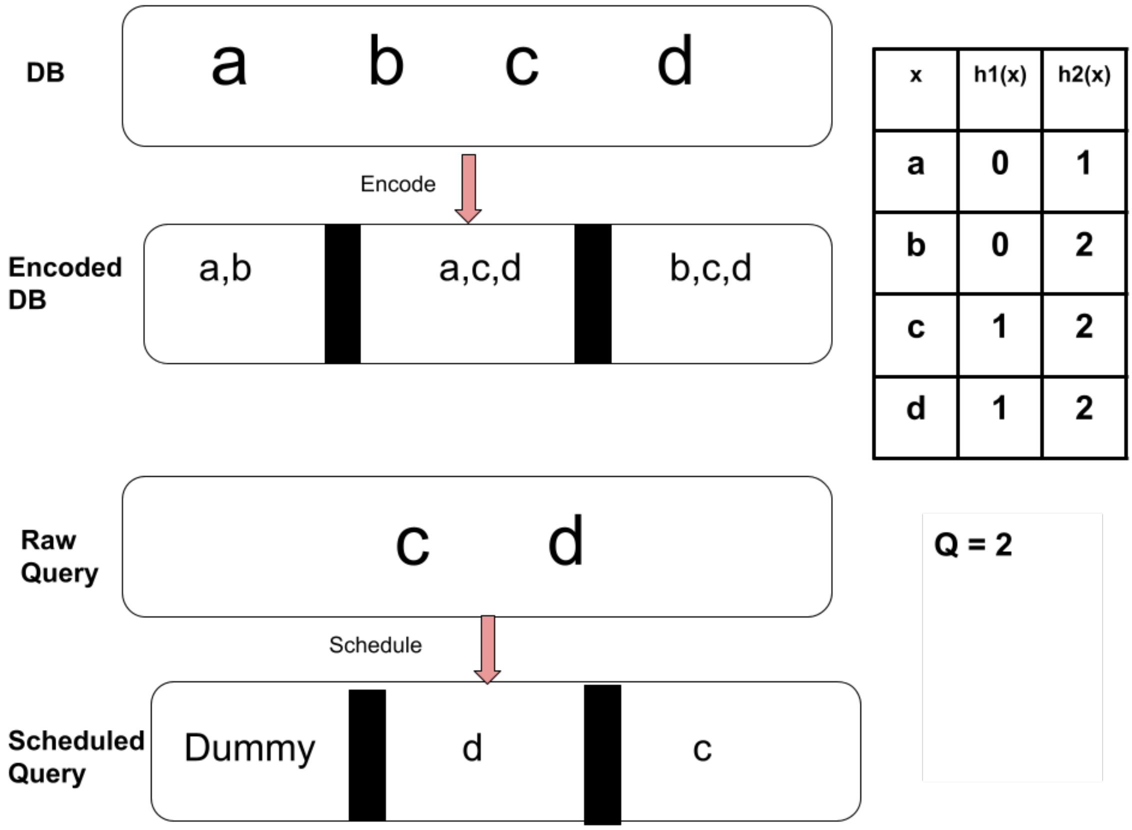


Figure 3: An example of the cuckoo hashing based batch PIR.