Cryptography Meets Algorithms (15893) Lecture Notes

Lecture 5: Piano: A Simple Preprocessing PIR Scheme

Scribe: Tanisha Saxena

February 16, 2024

1 Motivation

Recall that in previous lectures, we talked about classical PIR where the server stores the original database (DB) unencoded and there's no preprocessing. Classical PIR requires the server to look through each element of the database, otherwise the server can deduce that the skipped elements are unimportant to the client. Beimel et. al [BIM00] proved that a server in classical PIR must have linear computation per query. Linear computation will unlikely scale for many real-world applications, e.g., private DNS and private web search. To avoid this linear computation barrier, more recent PIR schemes considered the preprocessing model [BIM00, CK20]. In this lecture, we will show how we can achieve sublinear computation per query in the preprocessing model.

2 Background

Previous works have considered main models of pre-processing:

- **Public preprocessing:** The server computes an encoding of the dabase DB. This preprocessing is shared globally by all clients.
- **Client-specific preprocessing:** The server performs a preprocessing protocol with each client. The client is stateful, i.e., each client maintains some local state called the **hint**.

The "doubly-efficient PIR" scheme Wei-Kai talked about in his guest lecture is in the public preprocessing model. In this lecture, we will describe a scheme called Piano by Zhou et al. [ZPSZ23] that uses the client-specific preprocessing model.

Piano achieves $O(\sqrt{n})$ server time and $O(\sqrt{n})$ bandwidth per query, assuming $\widetilde{O}(\sqrt{n})$ client space. The only cryptographic primitive Piano relies is pseudorandom functions (PRFs).

3 Warmup Scheme: Supporting A Single Query

As a warmup, we will show a scheme that supports only one query after the preprocessing.

Given a database DB, we split the n indices into $\sqrt{n}$ chunks of size $\sqrt{n}$. We assume n is a perfect square.

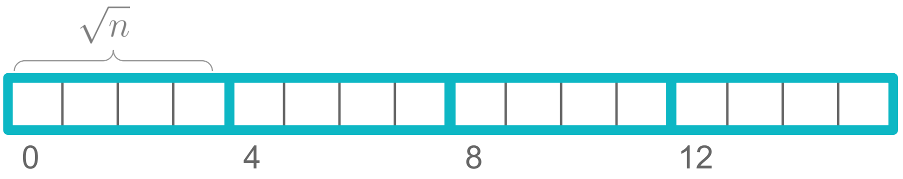

Figure: The database indices are divided into $\sqrt{n}$ chunks of size $\sqrt{n}$.

Random set. We will sample a random set of size $\sqrt{n}$ by sampling one random index from each of the $\sqrt{n}$ chunks, as illustrated in the picture below.

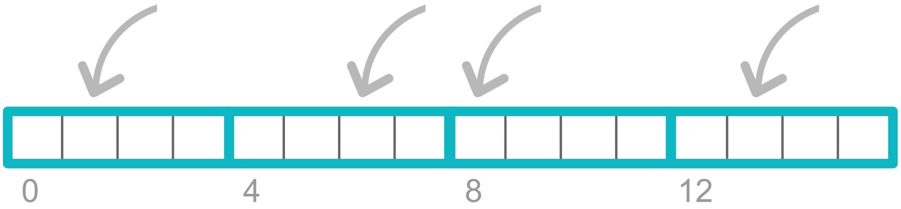

Figure: We sample a random set of size $\sqrt{n}$ by sampling one random index per chunk.

Given a random set S sampled according to the above procedure, we define the notation

$$\mathsf{Parity}(S) = \oplus_{i \in S} \mathsf{DB}[i]$$

Describing a set with a single PRF key. We need a succinct way to represent each set sampled as above. We can represent each set using a PRF key sk. Supposewe have a pseudorandom function family $\mathsf{PRF} : \{0,1\}^\lambda \times \{0, \ldots, \sqrt{n}-1\} \rightarrow \{0, \ldots, \sqrt{n}-1\}$. Given a PRF key $\mathsf{sk} \in \{0,1\}^\lambda$, we can compute a pseudorandom index from each chunk as follows: the index from the the i-th chunk is $\mathsf{PRF}_{\mathsf{sk}}(i) + i \cdot \sqrt{n}$ where $i \in \{0, \ldots, \sqrt{n}-1\}$. In other words, $\mathsf{PRF}_{\mathsf{sk}}(i)$ outputs the offset within the i-th chunk.

Henceforth, we use the notation $\mathsf{Set}(\mathsf{sk})$ to denote the pseudorandom set generated by the PRF key $\mathsf{sk} \in \{0,1\}^\lambda$. Given an index $j \in \{0, \ldots, \sqrt{n}-1\}$, we can determine if $j \in \mathsf{Set}(\mathsf{sk})$ in $O_\lambda(1)$ time, by checking if $\mathsf{PRF}_{\mathsf{sk}}(i) + i\sqrt{n} = j$ where $i = \mathsf{chunk}(j)$ denotes the chunk that index j belongs to.

Preprocessing. The client wants to prepare and store the following information during the preprocessing:

- **Hint table.** First, the client samples $L = \widetilde{O}(\sqrt{n})$ pseudorandom sets[1] represented by PRF keys $\mathsf{sk}_1, \ldots, \mathsf{sk}_L$. Further, for each $i \in [L]$, the client will find out the parity $p_i = \mathsf{Parity}(\mathsf{Set}(\mathsf{sk}_i))$, and store the parity bit alongside sk_i. The resulting table $\{(\mathsf{sk}_i, p_i)\}_{i \in [L]}$ stored by the client is called the hint table. Each entry (sk_i, p_i) in the hint table is called a hint.

- **Replacement entries.** For each chunk $i \in \{0, \ldots, \sqrt{n}-1\}$, the client samples $\widetilde{O}(1)$ random indices within the chunk; moreover, for each index r sampled, it wants to learn $\mathsf{DB}[r]$. As a result, the client will store $\widetilde{O}(1)$ replacement entries of the form $(r, \mathsf{DB}[r])$ for each chunk.

It takes $\widetilde{O}_\lambda(\sqrt{n})$ space to store the hint table and the replacement entries.

Claim 1. *The client can find out all L parities as well as the database at the sampled indices (for the replacement entries) by making a streaming pass over the database, consuming only $\widetilde{O}_\lambda(\sqrt{n})$ space.*

How to accomplish the above with a single streaming pass is left as an exercise.

[1] We use $\widetilde{O}(\cdot)$ to hide a superlogarithmic factor.

Making a query. Suppose the client wants to read index $q \in \{0, 1, \ldots, n-1\}$. It will do the following:

- Find a hint (sk^*, p^*) from its hint table such that $q \in \mathsf{Set}(\mathsf{sk}^*)$. If not found (which happens with negligible probability due to Claim 2), simply sample a random sk^* subject to $q \in \mathsf{Set}(\mathsf{sk}^*)$ and let $p^* = 0$.
- Let $i = \mathsf{chunk}(q)$, obtain the next unconsumed replacement entry belonging to the i-th chunk, denoted $(r, \mathsf{DB}[r])$, where index r belongs to the i-th chunk.
- Let $S = \mathsf{Set}(\mathsf{sk}^*)$ but replace the query q with r. Send S to the server.
- Wait for the server to return $p = \mathsf{parity}(S)$. Reconstruct the answer $p \oplus p^* \oplus \mathsf{DB}[r]$ where the client knows $\mathsf{DB}[r]$ from the replacement entry consumed.

Claim 2. *As long as the hint table has $\widetilde{O}(\sqrt{n})$ entries, the client can find a hint that matches the query except with negligible (in n) probability.*

Proof. Assume that the PRF is a random function — this will only affect the probability by a negligible amount. The probability that the query q is in a single set is $1/\sqrt{n}$. The probability that q is not in any of the L sets is $(1 - \frac{1}{\sqrt{n}})^L$. For $L = \omega(\sqrt{n} \log n)$, we have $(1 - \frac{1}{\sqrt{n}})^L = \left((1 - \frac{1}{\sqrt{n}})^{\sqrt{n}}\right)^{\omega(\log n)} = e^{-\omega(\log n)}$ which is negligible in n. □

Checking correctness of the answer is easy and left as an exercise. One can also easily verify that the scheme achieves the performance bounds claimed earlier.

Security of one query. Observe that the set S sent to the server is indisitinguishable from a randomly sampled set (i.e., sampling one random index per chunk). In particular, the client first searches for a set subject to containing the query q. If the PRF were a random function, then this set would have the same distribution as "sampling a random set subject to containing q". When we replace q with a random index from the same chunk, the resulting set would have the same distribution as a randomly sampled set.

4 Next Step: Supporting a Bounded Number of Random, Distinct Queries

The scheme so far can support one query, but an issue arises if the client continues to make more queries.

The question is, once we make a query, what happens to the hint that is consumed? If we keep the hint, then the next time the same hint is consumed, e.g., due to a different query, the server can learn the query index q that was scooped out earlier. If we simply remove the hint, it skews the distribution of the sets in the hint table. Specifically, the remaining sets have slightly lower probability of containing the query q. This will also cause slight information leakage in future queries.

To avoid information leakage over multiple queries, the idea is the following. Imagine that the PRF were a random function. Recall that the consumed set has the same distribution as "a randomly sampled set subject to containing the query q". When it is consumed, we want to replace it with a freshly sampled set subject to containing q — of course, we will also need to know the parity of this new set to maintain the correctness of the scheme.

To achieve this, the client will also prepare some backup sets as mentioned below.

Backup sets. During preprocessing, for each chunk, the client will prepare $\widetilde{O}(1)$ number of backup sets. A backup set for a chunk $i \in \{0, \ldots, \sqrt{n} - 1\}$ is sampled as follows: sample one random index for every chunk $j \neq i$. One can imagine that a backup set a random set except for leaving a hole in chunk i. Each backup set (for some chunk i) can also be described with a PRF key sk, except that we will never have to evaluate $\mathsf{PRF}_{\mathsf{sk}}(i)$.

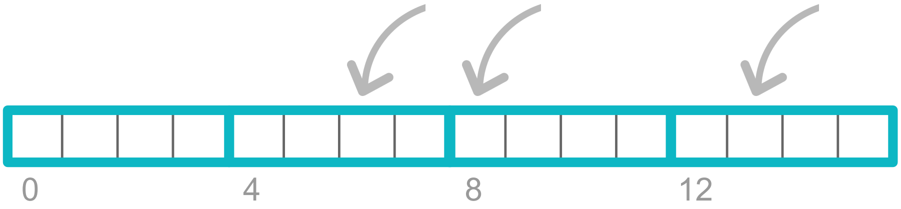

Figure: A backup set for chunk 0 contains a random index per chunk except for chunk 0.

The client also wants to learn and store the parities of all backup sets during preprocessing. This can be accomplished in the same streaming pass mentioned above.

Refreshing a consumed hint. During a query for index q, suppose the client consumes some hint. It will fetch the next unconsumed backup set for chunk $i = \mathsf{chunk}(q)$. along with its parity denoted (sk', p'). It will replace the consumed hint with $(\mathsf{sk}' \cup \{q\}, p' \oplus \mathsf{DB}[q])$, where $\mathsf{sk}' \cup \{q\}$ is used to mean "fill in the hole in the backup set represented by sk' with q"; further, $p' \oplus \mathsf{DB}[q]$ is the correct parity of the set when we fill in the hole with q. The client knows $\mathsf{DB}[q]$ because it has just reconstructed the answer to the present query.

Note that now in this scheme, each hint can be of the form (sk, p), or $(\mathsf{sk} \cup \{q\}, p)$, depending on whether it has been refreshed or not.

Why bounded, random and distinct queries. The scheme so far will work as long as we do not exhaust the replacement entries and the backup sets.

Suppose there are at most $\sqrt{n}$ queries after the proprocessing, and moreover, all queries are random and distinct. We can think of the $\sqrt{n}$ chunks as $\sqrt{n}$ bins, and each query will land in a random bin. By Chernoff bound, except with negligible probability, no bin will receive more than super-logarithmically many queries. This is why we provisioned $\widetilde{O}(1)$ many (i.e., superlogarithmically many) replacement entries and backup sets per chunk during preprocessing.

5 Supporting Unbounded and Arbitrary Queries

Given a bounded scheme supporting $\sqrt{n}$ queries, we can make it unbounded using a simple pipelining idea: i.e., we can perform the next phase's preprocessing in the current phase of $\sqrt{n}$ queries — specifically, this work can be spread equally among the queries of the current phase.

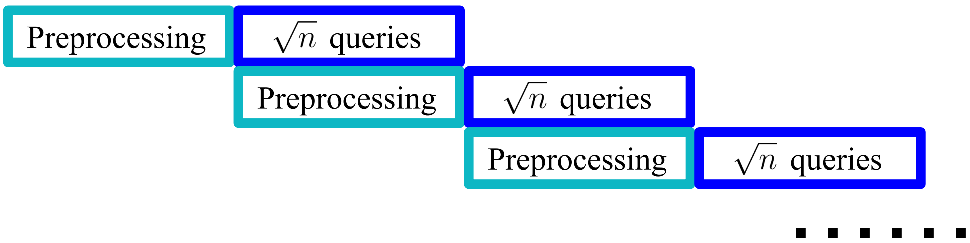

Figure: use pipelining to support an unbounded number of queries.

The random query assumption is needed only for load balancing, such that we do not run out of replacement entries and backup sets during each phase. We can get rid of the random query assumption by applying a public random permutation π to the database indices upfront. As long as the query generation process is independent of the choice of π, the load balancing guarantees still hold. In practice, we can use a pseudorandom permutation (PRP) instead of a completely random permutation π.

Finally, to get rid of the distinctness assumption, we can rely on the following simple idea. The client caches the answers to all queries in the current phase, which requires only $O(\sqrt{n})$ space. If a query is a duplicate copy of some query already made in the current phase, the client simply looks up the answer locally, and it instead issues a random query distinct from all queries in the current phase.

6 Concluding Remarks

Note that it may initially come off as surprising that we can get a scheme like Piano using only PRFs — recall that the existence of PRFs is equivalent to the existence of one-way functions. In an earlier lecture, we learned that classical PIR with sublinear bandwidth implies oblivious transfer, which cannot be constructed in a blackbox manner from one-way functions. One reason why we can get such a result is because Piano allows preprocessing phase with linear bandwidth. However, the cost of the preprocessing can be amortized over future queries.

References

[BIM00] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: Pir with preprocessing. In *Advances in Cryptology—CRYPTO 2000: 20th Annual International Cryptology Conference Santa Barbara, California, USA, August 20–24, 2000 Proceedings 20*, pages 55–73. Springer, 2000.

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

[ZPSZ23] Mingxun Zhou, Andrew Park, Elaine Shi, and Wenting Zheng. Piano: Extremely simple, single-server pir with sublinear server computation. *Cryptology ePrint Archive*, 2023.