Cryptography meets algorithms (15893) Lecture Notes

Lecture 5: PIR → OT and Preprocessing PIR

Scribe: Tanisha Saxena

February 6, 2024

1 Motivation

Recall that in last lecture, we talked about PIR where the server stores the original database (DB) unencoded and there's no preprocessing. This is incredibly inefficient, because it requires the server to look through each element of the database otherwise the adversary can deduce that the skipped elements are unimportant to the client. This was proven formally by Beimel et. al in 2000. [BIM00], where they prove that a server in classical PIR must have linear computation per query, with respect to the database. Following this discovery, Beimel et. al proved that preprocessing can overcome this complexity barrier. In this lecture, we'll understand how they did that.

2 Background

As we learned from guest lecturer Wei-Kai Lin, there are two main types of pre-processing.

Definition 1 (Types of Preprocessing in PIR)**.**

- ***Public Preprocessing:*** *Where the server encodes* DB *globally and maintains the same encoding scheme for all clients.*
- ***Client-Specific Preprocessing:*** *Where the server performs a preprocessing protocol with each client. The client is stateful, meaning each client maintains a local state called the* ***hint*** *referencing the preprocessing protocol.*

For this lecture, we'll be using client-specific preprocessing to achieve faster than linear time complexity.

3 PIANO: An Extremely Simple PIR

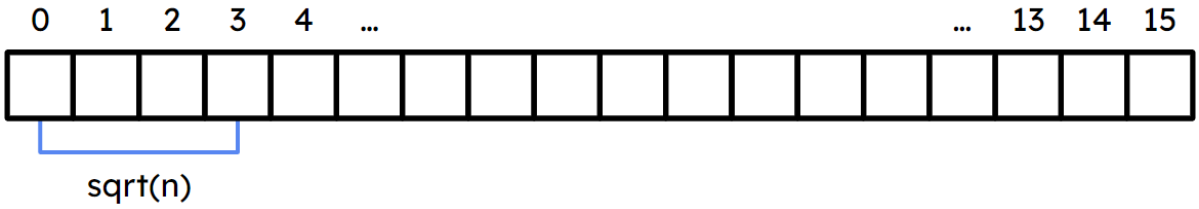
3.1 Goals

This PIR scheme by Zhou et. al achieves $O(\sqrt{n})$ time per query and $O(\sqrt{n})$ bandwidth [ZPSZ23] assuming $\tilde{O}(\sqrt{n})$ client space. The scheme assumes that pseudorandom functions (PRF) exists (One-way Function).

For the sake of our first pass, we're going to assume each query is random (i.e. the index of each message the client wants is randomly chosen), and that there are up to $\sqrt{n}$ unique queries. Basically, we're assuming our current queries are bounded and random. We'll remove this restriction later and are only using it now for the sake of ease.

3.2 The Scheme

Given a database DB, we split the n values into $\sqrt{n}$ chunks of size $\sqrt{n}$.



Definition 2.
Parity: *Given a set of indices* $S = \{i_1, i_2, \ldots, i_{\sqrt{n}}\}$*, we define* $\mathsf{Parity}(S)$ *as* $\mathsf{DB}(i_1) \oplus \mathsf{DB}(i_2) \oplus \cdots \oplus \mathsf{DB}(i_{\sqrt{n}})$

In the preprocessing phase, the client will randomly sample roughly $\Theta(\sqrt{n}\log n)$ random sets. For each set, it samples one random index in each chunk. The cilent will then preprocess all the parities for those sets. The preprocessing is done by streamingly download the whole database and dynamically update all the parities.

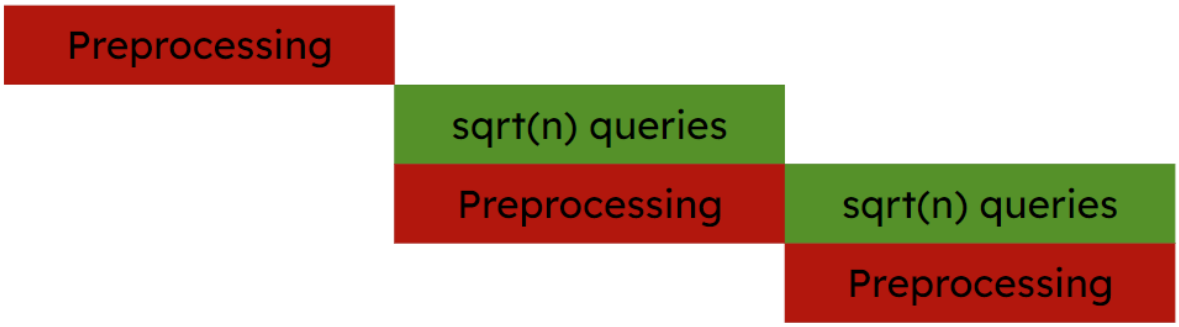
Compressing client hints with PRFs. Naively, to store all the hints, it takes $\Omega(n \log n)$ space. Assume we have a $\mathsf{PRF} : \{0,1\}^\lambda \times \{0, \ldots, \sqrt{n}-1\} \to \{0, \ldots, \sqrt{n}-1\}$, associated some key k. We can use this PRF to construct a random set. We define the element in the i-th chunk to be $\mathsf{PRF}_k(i) + i \cdot \sqrt{n}$. So each set only takes λ-bit to describe, and the client space is $O_\lambda(\sqrt{n}\log n)$.

Making Queries. A client makes queries in the following way. Consider a client that wants to query on index $i = 6$. Suppose the index we want to query is stored in some set $S_2 = (0, 6, 9, 5)$ – note that this set can be found amongst the $\sqrt{n}$ sets with non-negligible probability. To query on this, we simply sample some random r and send $S^* = (0, r, 9, 5)$ to the server. In return, we get $\mathsf{Parity}(S^*)$. With the help of our hint, we can compute $\mathsf{Parity}(S_2) \oplus \mathsf{Parity}(S^*) \oplus \mathsf{DB}[r] = \mathsf{DB}[i]$.

However, this raises a new issue. Suppose the new index we want to query is also in S_2. Then if we were to send an S^{**} with a different value replaced with r, the server would be able to recognize that S^{**} is similar to S^* and would then deduce some information about i. We also can't just delete S_2 to avoid this issue, because this would skew the distribution of the sets and would still give the server information about the query. The solution is that we use "backup" sets to replace the ones we've already used.

Backup Sets Alongside the hint, each client will store a polylog number of "backup" sets for each of the $\sqrt{n}\log n$ sets. Each backup set for chunk j is stored assuming the jth index is missing, and the client simply stores the parity of the backup set. Examples include: $\mathsf{Parity}(0, \_\_, 9, 5)$ and $\mathsf{Parity}(0, \_\_, 8, 13)$ for the second chunk.

Removing the "Bounded" and "Random" Queries Assumptions Note that within each set of $\sqrt{n}$ queries, the client can cache the results of the last $\sqrt{n}$ queries. Whenever it has a repeated query, it can make a dummy query and find the cached result. When we do the queries for the first $\sqrt{n}$ queries, we can simultaneously do the preprocessing to prepare for the next $\sqrt{n}$ queries.



And as easily as that, we have removed the bounded and random queries restrictions!

Applications This scheme is much faster than non-processing PIR schemes. Because of that, it has many applications for fast secrecy. For example, haveibeenpwned.com, a website that checks if a user's password has been in a data breach, would be able to more efficiently run a search on a user's password without actually ever knowing what the password is.

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.

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