Cryptography meets algorithms (15893) Lecture Notes

Lecture 5: PIR $\rightarrow$ OT and Preprocessing PIR

Scribe: Tanisha Saxena

February 14, 2024

1 PIANO: An Extremely Simple PIR

1.1 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.

1.2 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**.

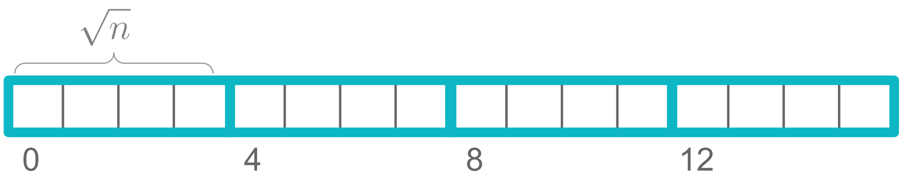
The "doubly-efficient PIR" scheme Wei-Kai talked about in his guest lecture is in the public preprocessing model. In this lecture, we will describe a scheme called Piano by Zhou et al. [ZPSZ23] that uses the client-specific preprocessing model.

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).

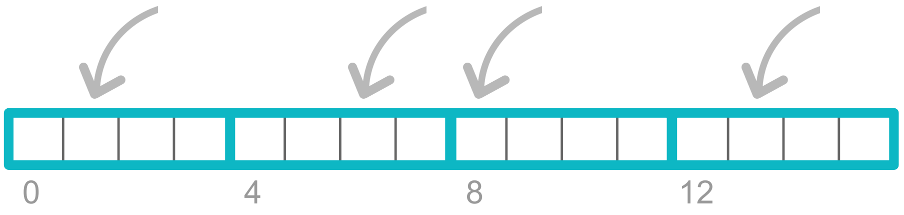
1.3 Warmup Scheme: Supporting A Single Query

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.



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.



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 learn 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 $\mathsf{DB}[r]$.

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.

Making Queries. Suppose the client wants to read index $q \in \{0, 1, \ldots, n-1\}$. It finds a hint (sk^*, p^*) from its hint table such that $q \in \mathsf{Set}(\mathsf{sk}^*)$.

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

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

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 $\mathsf{Parity}(S_2) \oplus \mathsf{Parity}(S^*) \oplus \mathsf{DB}[r] = \mathsf{DB}[i]$.

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.

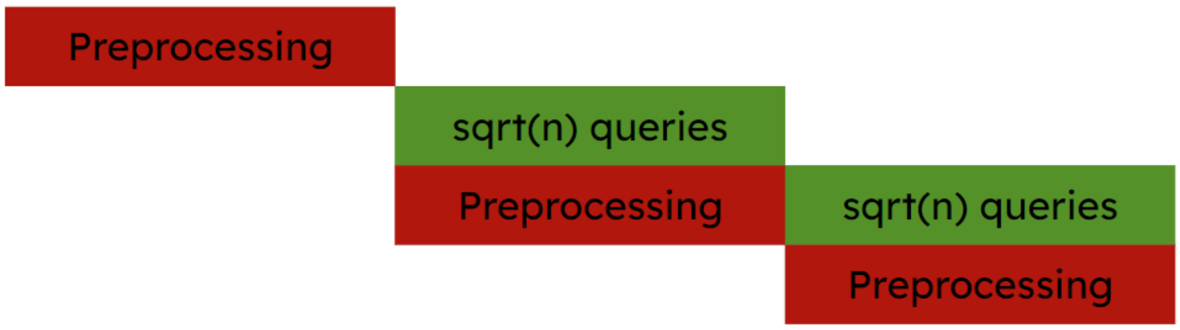
Compressing client hints with PRFs. Naively, to store all the hints, it takes $\Omega(n \log n)$ space. Assume we have a $\mathsf{PRF} : \{0,1\}^\lambda \times \{0, \ldots, \sqrt{n}-1\} \to \{0, \ldots, \sqrt{n}-1\}$, associated some key k. We can use this PRF to construct a random set. We define the 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.

[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.