Function Secret Sharing and PIR

Shamir secret sharing allows a dealer to split a value across many parties.

- possible to perform arbitrary computation on secret-shared data.
- each user secret-shares its input (addition is local, multiplication requires communication, degree-reduction).
- very efficient; information-theoretic (honest-majority for semi-honest security).

Function secret sharing [Boyle, Gilboa, Ishai '15]: allows a dealer to split a function.

- guarantee: for any $f$:
  \[ \sum_{i=1}^{n} f_i(x) = f(x) \]

- requirements: function shares should be:
  - succinct (otherwise, can have trivial construction where truth table is secret-shared).
  - not reveal anything about the function $f$ (information-theoretically: secret share truth table, but more efficient construction possible with computational hiding properties).

This lecture: consider one special case of function secret-sharing (for distributed point functions (DPFs):

- Introduced by Gilboa and Ishai (Crypto 2014) - surprisingly powerful and useful primitive.
- Point function: $f_g(x) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$.
- Distributed point function consists of two algorithms (Gen and Eval):
  
  \[ \text{Gen}(y) \rightarrow (k_0, k_1) \rightarrow \text{generates keys for point function at } y \]
  
  \[ \text{Eval}(k_0, x') \rightarrow y' \rightarrow \text{evaluates point function at } x' \]

- Correctness: for all points $x, y, f$: $(k_0, k_1) \leftarrow \text{Gen}(y)$;
  \[ \text{Eval}(k_0, x) \oplus \text{Eval}(k_1, x) = f_g(x) \]

- Security: for all points $y$: $(k_0, k_1) \leftarrow \text{Gen}(y)$;
  \[ \{ k_b \}_{b \in \{0, 1\}} \approx \text{Sim}(b, k_0, k_1, y) \]

Intuitively: key $k_b$ reveals nothing about point $y$, other than size of domain and range.
Why PIRs?

Gives an immediate solution for multi-server PIR (private information retrieval) for reading: databases are replicated on multiple servers

![Diagram of two servers and a client](image)

- wants to read record i without revealing index i to the servers

Applications: perform queries to a database without revealing query to server

- private flight bookings to prevent discriminatory pricing
- Splinter system [NSDI 2017]
- private navigation to ensure location privacy
- private lookups in Tor hidden services

Can also consider reverse problem: writing to a database without revealing which position was updated

- very useful for anonymous messaging: Riporta system [Oakland 2015]

Closely related to oblivious transfer (no requirement for sender privacy, only receiver privacy)

- goal in PIR is to minimize communication (in OT, usually it's to minimize computation, but can combine OT with PIR to reduce communication — "strong PIR")

Information-Theoretic PIR

Two-server PIR with \(O(\sqrt{n})\) communication:

- View databases as \(\sqrt{n}\)-by-\(\sqrt{n}\) matrices:

  ![Diagram of two matrices and a client](image)

  - \(Z_1 = D \cdot V_1\)
  - \(Z_2 = D \cdot V_2\)
  - \(z_1 + z_2 = D(V_1 + V_2) = D v_i\)

- Can reduce communication to \(O(n^{1/3})\) [CGKS98]

- Best known lower bound: \(\log n\) (trivial lower bound is \(\log n\) - need to communciate bits of index)

- Conjectured lower bound \(\Omega(n^{1/2})\) in [CGKS98]

- Breakthrough work by Dvir, Gopi (2015): 2-server PIR with communication \(O(\sqrt{\log n \log \log n})\)
Distributed Point Functions (DPFs)

Relies on computational assumptions and gives 2-server PIR with polylog communication

- Note: with computational assumptions, can have single-server PIR with polylog communication but this requires algebra (in fact, single-server PIR with sublinear communication implies OT, so algebra is probably necessary)
- DPFs give very efficient construction in 2-party setting (relying only on one-way functions)

2-server PIR from DPF:

Server 1

\[ V_i : \overline{z}_i \leftarrow \text{Eval}(k_i) \]

\[ \overline{z}_i^{(0)} = \sum_{j \in \mathbb{F}_m \setminus \{0\}} \overline{D}_j \overline{z}_j^{(0)} \]

\[ \overline{z}_i^{(1)} \leftarrow \text{Eval}(k_i) \]

\[ \overline{z}_i^{(1)} = \sum_{j \in \mathbb{F}_m \setminus \{0\}} \overline{D}_j \overline{z}_j^{(1)} \]

Client (i)

\( (k_1, k_2) \leftarrow \text{Gen}(i) \)

\[ \overline{z}_i^{(0)} + \overline{z}_i^{(1)} = \sum_{j \in \mathbb{F}_m \setminus \{0\}} \overline{D}_j \left( \overline{z}_j^{(0)} + \overline{z}_j^{(1)} \right) = 0 \text{ except } i = j \]

Total communication: \[ |k_1| + |k_2| + |\overline{z}_i^{(0)}| + |\overline{z}_i^{(1)}| \]

\[ O(\log n), \quad O(1) \text{ for bits} \]

2-server Writable PIR from DPF:

Server 1

\[ D_i^{(0)} \leftarrow D_i^{(0)} + \text{Eval}(k_i) \]

\[ D_i^{(1)} \leftarrow D_i^{(0)} + \text{Eval}(k_i) \]

Server 2

Database is secret shared:

\[ D = D_i^{(0)} + D_i^{(1)} \]

(initially all zeroes)

Client (i)

\( (k_1, k_2) \leftarrow \text{Gen}(i) \)

DPF guarantees that update only flips bit at position i

Take-away:
Reading obliviously from a database: secret-share query
Writing obliviously to a database: secret-share query
Distributed Point Functions

- Start by constructing a DPF with \( \sqrt{n} \)-size keys (where the domain size is \( n \)).
- View output of DPF as \( \sqrt{n} \)-by-\( \sqrt{n} \) grid — compress using PRG.

\[
\begin{array}{|c|c|c|}
\hline
s_1 & s_2 & s_3 \\
\hline
s_1' & s_2' & s_3' \\
\hline
s_2 & s_3 & s_4 \\
\hline
s_2' & s_3' & s_4' \\
\hline
\end{array}
\]

two functions differ only at shaded index

1. Associate random PRG seed with each column 
   \( \Rightarrow \) use PRG to derive pseudorandom string for each row 
   \( \Rightarrow \) different PRG for the target row
   \( \Rightarrow \) but row copy element in the row differs

2. Introduce a correction factor for the columns

\[
\begin{array}{|c|c|c|}
\hline
b_1 & s_1 & G(s_1) \oplus b_1 \\
\hline
b_2 & s_2 & G(s_2) \oplus b_2 \\
\hline
b_3 & s_3 & G(s_3) \oplus b_3 \\
\hline
b_4 & s_4 & G(s_4) \oplus b_4 \\
\hline
\end{array}
\]

Correctness: row other than \( i \) are identical 
row \( j \) xors to \( e_j \) by construction

Security: seeds and control bits uniformly random, correction factor blinded
   by \( G(s_i') \), so keys computationally indistinguishable from random

Efficiency: \( \sqrt{n} \)-size keys

\( (99\%) \)

Towards Logarithmic-Sized Keys:

Observation: keys in \( \sqrt{n} \)-DPF have following structure

\[
\begin{align*}
\sqrt{n} & \left\{ \begin{array}{c}
(s_i, b_i, v_i) \\
(s_i, b_i, v_i) \\
(s_i, b_i, v_i) \\
(s_i, b_i, v_i) \\
\end{array} \right. \\
\end{align*}
\]

- Can also build iteratively using tree-based construction
- Lots of applications (very practical!)

\( \Rightarrow \) can be viewed as shares of a point function over domain of size \( \sqrt{n} \) (for some special point)
\( \Rightarrow \) compress using another DPF on \( \sqrt{n} \) recursively apply construction to obtain \( \sqrt{n} \)-by-\( \sqrt{n} \) size