CS 65500
Advanced Cryptography

Lecture 19: Function Secret Sharing

Instructor: Aarushi Goel

Spring 2025

Agenda

→ Function Secret Sharing

→ Motivation

→ Distributed Point Function

Secret Sharing Inputs vs Functions

→ So far, we have seen how to a dealer can secret share an input amongst mutually distrusting parties.

→ When using a linear secret sharing scheme, given secret shares of inputs, parties can non-interactively obtain secret shares of any linear combinations of the inputs.

→ Today we are going to discuss a dual notion of *function secret sharing* (FSS)

→ FSS enables a dealer to secret share a function amongst mutually distrusting parties

→ Given secret shares of the function, the parties should be able to non-interactively compute shares of the output of this function on any common input.

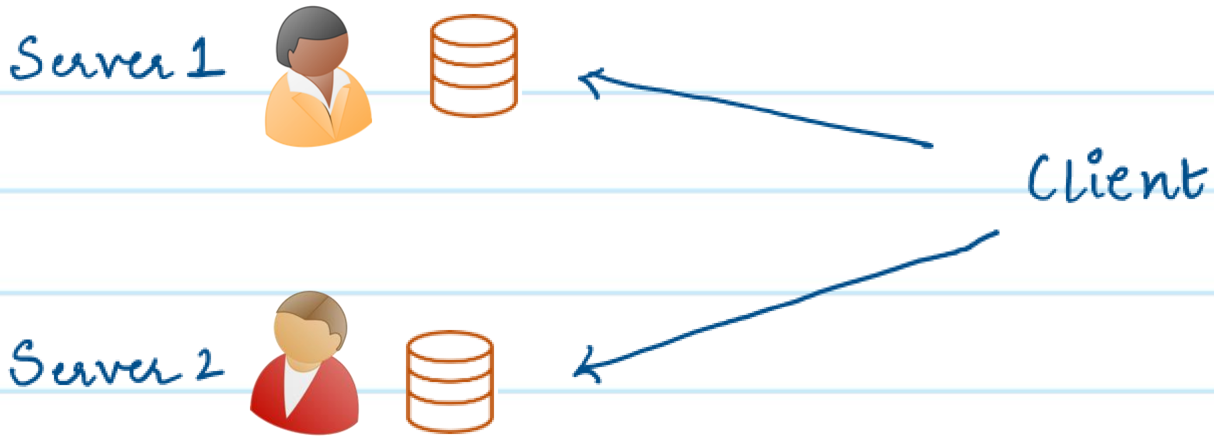
Function Secret Sharing

$$
\begin{tikzcd}[column sep=large]
 & f_1 \arrow[r, "\mathrm{Eval}(f_1,\, x)"] & f_1(x) \\
f \arrow[ur] \arrow[dr] & & \\
 & f_2 \arrow[r, "\mathrm{Eval}(f_2,\, x)"] & f_2(x)
\end{tikzcd}
\qquad f_1(x) + f_2(x) = f(x)
$$

Share   Evaluation

- → Function f is secret shared, input x is common.
- → Each f_i should hide necessary information about f.
- → Trivial Solution: Additive secret share each entry in the truth table of f.
  However this will result in shares that are linear in the domain size
- → Challenge: Design FSS where shares are sublinear in the domain size.

Private Database Queries



Let's assume both servers hold a copy of a database. The client wants to query this database without revealing the query attributes (but revealing the query structure is okay!)

Example: Database contains the list of all the restaurants in west Lafayette. Client wants to get a COUNT of *Vegetarian*, *Greek* restaurants in the city.

ok with revealing   should remain hidden

Query: COUNT (column) where $x_1 = v_1$, $x_2 = v_2$

vegetarian Greek

→ We can define a predicate f_{y_1,y_2} such that $f_{y_1,y_2}(x_1, x_2) = 1$ iff $x_1 = y_1$ and $x_2 = y_2$, and 0 otherwise.

→ The client can use FSS to compute shares of f_{y_1,y_2} and send them to the servers.

→ The servers can use FSS shares and evaluate on all entries in the database. Sum these evaluations $z_i = \sum_j f_i(x_1^j, x_2^j)$ & send the sum to the client.

→ Client computes: $z_1 + z_2 = \sum_j f_{y_1,y_2}(x_1^j, x_2^j)$.

A similar approach can be used to compute other statistical queries.

Function Secret Sharing for a Point Function

→ Point Functions are of the form:

$$f_{(v_1,\dots,v_n)}(x_1,\dots,x_n) = \begin{cases} 1, & \text{if } x_1 = v_1, \dots, x_n = v_n \\ 0, & \text{otherwise} \end{cases}$$

we can simply this to:

$$f_y(x) = \begin{cases} 1, & \text{if } x = y \\ 0, & \text{otherwise} \end{cases}$$

→ FSS for point functions is called a *distributed point function** (DPF)

→ DPFs are very useful in various secure computation applications.

(we will see one such application in the next class)

Defining Two-Party Distributed Point Function

Let $f:\{0,1\}^n \to \mathbb{G}$ be a point function.

Definition: A two-party DPF scheme is defined by PPT algorithms (Gen, Eval):

* $\mathrm{Gen}(1^\lambda, f)$: On input f and the security parameter 1^λ, it outputs shares f_1, f_2.
* $\mathrm{Eval}(i, f_i, x)$: On input $i \in [2]$, share f_i and input $x \in \{0,1\}^n$, it outputs $y_i \in \mathbb{G}$.

These algorithms must satisfy the following properties:

→ Correctness: For any point function $f:\{0,1\}^n \to \mathbb{G}$ and any input $x \in \{0,1\}^n$, if $f_1, f_2 \leftarrow \mathrm{Gen}(1^\lambda, f)$, then $\Pr[\mathrm{Eval}(1, f_1, x) + \mathrm{Eval}(2, f_2, x) = f(x)] = 1$

→ Security: For any point functions g, h, and any $i \in [2]$, the following distributions are computationally indistinguishable:

$$\{ g_i \mid (g_1, g_2) \leftarrow \mathrm{Gen}(1^\lambda, g) \}$$

$$\{ h_i \mid (h_1, h_2) \leftarrow \mathrm{Gen}(1^\lambda, h) \}$$

Construction of DPFs.

→ We can build 2-party DPFs from OWFs.

→ Most efficient known construction of 2-party DPFs yields shares of size polylog in the domain size.

→ Today: We will discuss a simpler construction where shares are of size square root in the domain size.

→ Both of these constructions are by:

Elette Boyle

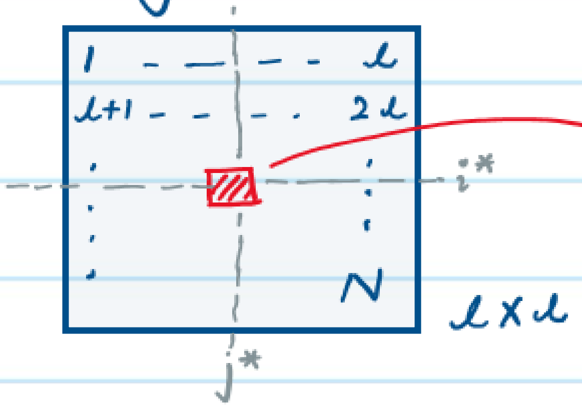
Niv Gilboa

Yuval Ishai

Construction of DPFs.

Let N be the size of the domain. For simplicity lets consider point functions of the form $f_y: [N] \to \{0,1\}$. Let $\ell = \sqrt{N}$

1. Let us start by listing all the domain elements in an $\ell \times \ell$ size matrix



Let the $(i^*, j^*)^{th}$ element in this matrix be the input on which $f_y()$ outputs 1.

(We know that $f_y()$ outputs 0 on all other inputs)

2. Let $PRG: \{0,1\}^\lambda \to \{0,1\}^\ell$ be a pseudorandom generator. We will sample a PRG seed s_i for each row $i \in [\ell]$.

3. Starting Idea: Each FSS share f_i will consist of all the seeds $s_1, \ldots, s_\ell$. For evaluating on any input x, determine its corresponding location (i,j) in the matrix, then compute $PRG(s_i)$ & output the j^{th} bit.

For each $i \neq i^*$: Observe that both parties will compute the same shares: $\text{Eval}(1, f_1, x) = \text{Eval}(2, f_2, x) = \text{PRG}(s_i)[j]$.

$\Rightarrow$ They compute shares of 0, which is what we want.

For row i^*: For the input corresponding to location (i^*, j^*) in the matrix, we want parties to be able to compute additive secret shares of 1, not 0. But for inputs corresponding to all other (i^*, j), $j \neq j^*$, we still want parties to be able to compute additive shares of 0.

$\Rightarrow$ The approach described on the previous slide does not work!

Modification:

- Share f_1 will consist of all seeds $s_1, \ldots, s_\ell$

- Share f_2 will consist of the same seeds except s_{i^*} will be replaced with a random independently sampled s'_{i^*}

- In addition to these seeds, both parties also get a correction word w, such that $PRG(s_{i^*}) \oplus PRG(s'_{i^*}) \oplus w = \boxed{e_{j^*}}$

  ℓ length bit string that is 0 everywhere except at position j^*.

- During evaluation, we want this correction word to be used only when $i = i^*$. But this must be done in a manner that does not reveal i^* to either party.

Final Construction:

→ Gen$(1^\lambda, f_y)$: Let the location corresponding to y in the matrix be (i^*, j^*)

$\forall i \in [l]$, sample $s_i \xleftarrow{\$} \{0,1\}^\lambda$, sample another $s'_{i^*} \xleftarrow{\$} \{0,1\}^\lambda$.

$\forall i \in [l]$, sample bits $b_1, \dots b_l \xleftarrow{\$} \{0,1\}$.

Compute: $w = PRG(s_{i^*}) \oplus PRG(s'_{i^*}) \oplus e_{j^*}$

* Share f_1: $(s_1, \dots, s_l, w, b_1, \dots, b_l)$.

* Share f_2: $(s_1, \dots s'_{i^*}, \dots, s_l, w, b_1, \dots \dots (1-b_{i^*}), \dots, b_l)$

→ Eval(i, f_i, x): Let (i,j) be the location corresponding to input x.

output $y_i = (PRG(s_i) \oplus b_i \cdot w)[j]$

Correctness: Depending on (i,j) corresponding to x, we consider the following cases:

1.) $i \neq i^*$: $y_1 = (PRG(s_i) \oplus b_i \cdot w)[j]$

$y_2 = (PRG(s_i) \oplus b_i \cdot w)[j]$

$y_1 \oplus y_2 = 0 = f_y(x)$

2.) $i = i^*, \quad j \neq j^*$: $y_1 = (PRG(s_{i^*}) \oplus b_{i^*} \cdot w)[j]$

$y_2 = (PRG(s'_{i^*}) \oplus (1-b_{i^*}) \cdot w)[j]$

$y_1 \oplus y_2 = (PRG(s_{i^*}) \oplus PRG(s'_{i^*}) \oplus w)[j] = e_{j^*}[j$

3.) $i = i^*, \quad j = j^*$: $y_1 = (PRG(s_{i^*}) \oplus b_{i^*} \cdot w)[j^*]$

$y_2 = (PRG(s'_{i^*}) \oplus (1-b_{i^*}) \cdot w)[j^*]$

$y_1 \oplus y_2 = (PRG(s_{i^*}) \oplus PRG(s'_{i^*}) \oplus w)[j^*] = e_{j^*}[j^*] = 1$

Security: We want to show that each share f_i, hides y (or equivalently i^*, j^*)

1. $f_1 = (\underbrace{s_1, \ldots, s_\ell}_{\text{uniform}}, w, \underbrace{b_1, \ldots, b_\ell}_{\text{uniform}})$ | i^* remains hidden.

$$w = PRG(s_{i^*}) \oplus \underbrace{PRG(s'_{i^*})} \oplus e_{j^*}$$

Since this party does not get s'_{i^*}, $PRG(s'_{i^*})$ acts as a one-time pad for masking e_{j^*}. Hence j^* remains hidden.

2. $f_2 = (s_1, \ldots, s_{i^*}, \ldots, s_\ell, w, b_1, \ldots, (1-b_{i^*}), \ldots b_\ell)$

similar argument as above