Distributed Point Functions Continued

CS 507, Topics in Cryptography: Secure Computation

David Heath

Fall 2025

Last time, we introduced the notion of a distribution point function (DPF), and we saw that it was useful for solving the two-server private information retrieval problem. Recall that a DPF is defined as follows:

Definition 1 (Two Party DPF)**.** *A DPF is a pair of algorithms* $\texttt{KeyGen}, \texttt{Eval}$*:*

- $\texttt{KeyGen}$ *takes as input a point* $\alpha \in [n]$ *and a value* $\beta \in \{0,1\}^w$*. It outputs two keys* k_0 *and* k_1*.*
- $\texttt{Eval}$ *takes as input a key* k *and an index* i*. It outputs a* w*-bit string.*

The DPF is ***correct*** *if the following holds for all* α*,* i*, and* β*:*

$$\Pr\left[\texttt{Eval}(k_0, i) \oplus \texttt{Eval}(k_1, i) = \begin{cases}\beta & \text{if } i = \alpha \\ 0 & \text{otherwise}\end{cases} \quad \text{where} \quad (k_0, k_1) \leftarrow \texttt{KeyGen}(\alpha, \beta)\right] = 1$$

As a reminder, the intuition behind a DPF is that it is an efficient way to secret share a *one-hot* vector. Last time, we claimed, without proof, that DPFs can be constructed where the keys have length only $O(\lambda \lg n)$ bits, assuming only that PRGs exist. Today, we will see how this construction works [BGI16].

Then, we will talk about the relevance of DPFs to MPC. In particular, we will see how DPFs naturally allow to solve a problem of generalizing Boolean gates to *lookup tables*.

DPFs from PRGs

To begin, let us assume that we have access to a PRG G that (slightly more than) doubles the length of its input:

$$G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda+2}$$

It will also be useful to parse the output of G into two halves; namely, we will define functions G_0, G_1 as follows:

$$G_0, G_1 : \{0,1\}^\lambda \to \{0,1\}^{\lambda+1}$$
$$G_0(x), G_1(x) = G(x)$$

That is $G_0(x)$ and $G_1(x)$ simply denote the two halves of the output of G.

Recall that a DPF allows a client to create two keys, one that will be held by each server. These keys allow the servers to locally reconstruct XOR secret shares. The DPF keys themselves will be built up in a recursive manner by using G to expand shares of increasing length. It will therefore be extremley useful to define what it *means* for the two servers to invoke a PRG G on a secret-shared string. Let $s \in \{0,1\}^\lambda$ be a string secret-shared between the servers. We define the following notation:

$$G([s]) = G((s0, s1)) \triangleq (G(s_0), G(s_1))$$

There are two crucial things worth observing about this notation:

- First, suppose that s is a unifom string that is not known to either party. In that case, $G([s])$ is *itself* a secret-shared uniform string $[r]$ where $r \in \{0,1\}^{2\lambda+2}$. The reason for this is that, for example, server S_0 does not know seed share s_1, and hence by PRG security $G(s_1)$ is uniform from S_0's perspective.

- Second, suppose that s is the all-zeros string. That is, $s_0 = s_1$. In that case, $G(s_0) = G(s_1)$, so the expression $G([0^\lambda])$ evaluates to $[0^{2\lambda+2}]$.

The central idea that enables efficient DPFs is that there is no mechanism by which a server can locally tell which of the two above cases it is in! Specifically, we can arrange that if s is uniform, then S_0 holds some uniformly random seed share s_0; but if s is all zeros, we can still arrange the same thing!

Now that we have this notation and observation about secret-shared seeds, we are ready to construct an efficient DPF. The procedure will proceed recursively. In particular, suppose the client wishes to construct a secret-shared one-hot vector of length n. To do so, the client will first arrange keys that allows servers to obtain a secret-shared one-hot vector of length 1, then use those keys to construct keys that yield a one-hot vector of length 2, then length 4, ..., until ultimately reaching a secret-shared vector of length n.

In slightly more detail, each intermediate value of length m will be a secret-shared *array* $[A]$ of length m, where each array slot A_i is a $(\lambda+1)$-bit string. λ bits of each slot are the *seed part*, and the last bit is its *indicator bit*. Let α denote the *active* slot of this one-hot array. We will arrange, inductively, that slot A_α holds (1) a pseudorandom string of length λ followed by (2) the indicator bit 1. All other slots $A_{i\neq\alpha}$ will hold the all zeros string.

In particular, if the client wishes to generate keys for a one-hot vector of length n with active position α, it first recursively generate keys for a one-hot vector of length $n/2$ with active position $\lfloor \alpha/2 \rfloor$. Suppose the servers hold such an array. That is, the hold secret shares of an array of the following form:

$$[0^\lambda 0, 0^\lambda 0, ..., 0^\lambda 0, s1, 0^\lambda 0, ..., 0^\lambda 0, 0^\lambda 0]$$

where $s \in \{0,1\}^\lambda$ is some pseudorandom string. Now, suppose that for each array slot $[A_i]$, the parties apply G_0 and G_0 to the secret-shared "seed part" of that index. There are two cases to consider:

$$G_0([0^\lambda]) = [0^\lambda 0] \qquad G_1([0^\lambda]) = [0^\lambda 0]$$
$$G_0([s]) \stackrel{c}{=} \{[r] \text{ where } r \in_\$ \{0,1\}^{\lambda+1}\} \qquad G_1([s]) \stackrel{c}{=} \{[r] \text{ where } r \in_\$ \{0,1\}^{\lambda+1}\}$$

If we now concatenate the results of all of these expansions, we obtain a secret-shared array A' with twice the length of A. Moreover A' is *almost* of the form of our invariant. A' is zero everywhere, except where it the true seed s was expanded. Specifically, A' looks like this:

$$[0^\lambda 0, 0^\lambda 0, 0^\lambda 0, 0^\lambda 0, ..., 0^\lambda 0, 0^\lambda 0, (R_0, r_0), (R_1, r_1), 0^\lambda 0, 0^\lambda 0, ..., 0^\lambda 0, 0^\lambda 0, 0^\lambda 0, 0^\lambda 0] \tag{1}$$

Here R_0, R_1 denote the seed parts, and r_0, r_1 denote the indicator bits. Here, the random strings (R_0, r_0) and (R_1, r_1) respectively reside in array positions $2\lfloor \alpha/2 \rfloor$ and $2\lfloor \alpha/2 \rfloor + 1$. To restore out invariant, the client must somehow "correct" these two positions. To do so, the client will send to the two servers the following *correction word*:

$$\texttt{cw} = \begin{cases} R_1, (r_0 \oplus 1), r_1 & \text{if } \alpha = 2\lfloor \alpha/2 \rfloor \\ R_0, r_0, (r_1 \oplus 1) & \text{if } \alpha = 2\lfloor \alpha/2 \rfloor + 1 \end{cases}$$

Upon receiving correction word $\texttt{cw} = (R, b, c)$, the server first expands it into a string $\texttt{cw}' = (R, b, R, c)$. This means that we wish to *cancel* the R_1, r_1 terms, and set the R_0, r_0 term to some random string followed by a 1.

To do this, the servers will for each i multiply their secret-shared indicator bit in from A_i by $\texttt{cw}'$. Because only the single active slot $A_{\lfloor \alpha/2 \rfloor}$ has an indicator bit of 1, this yields the secret-shared array:

$$[0^\lambda 0, 0^\lambda 0, 0^\lambda 0, 0^\lambda 0, ..., 0^\lambda 0, 0^\lambda 0, (R, b), (R, c), 0^\lambda 0, 0^\lambda 0, ..., 0^\lambda 0, 0^\lambda 0, 0^\lambda 0, 0^\lambda 0] \tag{2}$$

To restore the invariant, the servers bitwise XOR the arrays in Equations (1) and (2). Correctness of this approach can be seen by the structure of `cw` and case analysis on whether α is even or odd. In either case, the inactive slot is set to all zeros, and the active slot is restored to a random string $R_0 \oplus R_1$, followed by a 1.

Thus, we are done: We managed to turn a secret-shared one-hot vector of length $n/2$ into one of length n, at cost of a correction word whose size is only $\lambda + 2$ bits. This ultimately gives a full `KeyGen` procedure: The client first includes in the keys a trivial shared one-hot array of length 1, the repeatedly doubles the secret shared array length by including a correction word of length $\lambda + 2$. After $\lg n$ doublings, the client is done. Hence, the full key-length is only $O(\lambda \lg n)$ bits.

Computational Efficiency. As described, the client/servers proceed by imagining the entire secret-shared arrays. However, if we revisit this, we can see that this is unnecessary. In particular, the client needs only keep in its head the "active" path through the sequence of one-hot encodings. Thus, the client computation cost includes only $O(\lg n)$ PRG evaluations. Similarly, if the servers wish to evaluate their keys at some particular position i, they can do so by following the path towards i, at cost $O(\lg n)$ PRG evaluations.

Secret-Shared Lookup Tables

We will now discuss utility of DPFs in the context of an MPC protocol. In particular, our goal will be to augment standard two-party GMW with support for arbitrary *lookup tables*. Suppose the parties agree on some function $f : \{0,1\}^n \to \{0,1\}$, and let us assume that n is very small (i.e. at most logarithmic in the security parameter). What we would like is to augment GMW with support for an efficient gate that computes f, rather than, say, AND or XOR.

We will see now how to do this in 1 round and $O(n)$ bits of online communication. To do this, suppose that the parties were given by a dealer a secret-shared one-hot encoding $[\mathtt{hot}(\alpha)]$, as well as $[\alpha]$ for some uniformly random $\alpha \in \{0,1\}^n$. Now, as GMW runs, they obtain some gate input $[x]$, and they wish to obtain $[f(x)]$.

To achieve this evaluation, the parties first locally compute $[x \oplus \alpha]$, then open $x \oplus \alpha$ to themselves. Note that this is safe, because α acts as a one-time pad. From here, the parties *rotate* the one-hot vector $[\mathtt{hot}(\alpha)]$ by distance $x \oplus \alpha$. Namely, they create a new vector h as follows:

$$[h_i] = [\mathtt{hot}(\alpha)]_{i \oplus x \oplus \alpha}$$

A bit of straightforward reasoning shows that h is equal to $\mathtt{hot}(x)$. Next, the parties locally compute the following:

$$\bigoplus_i f(i) \cdot [\mathtt{hot}(x)]$$

Essentially, this is operation corresponds to taking the inner product of the one-hot vector with the *truth table* for f. Again, a bit of straightforward reasoning shows that the result of this computation is $[f(x)]$. So we are done!

We have successfully computed $[f(x)]$ for *arbitrary* function f, an all that was required was to reveal some difference $x \oplus \alpha$ of length n.

Now, we can see the role of DPFs in this setting: for this to work, we need a long, secret-shared one-hot vector, and we know that such vectors can be compactly represented as short keys.

Distributed Key Generation for DPFs. To use DPFs in the context of an MPC protocol, we need the keys to be generated by some dealer. In the past, we have replaced the dealer with another preprocessing protocol. Can we do the same with DPFs? Indeed, in the two-party setting, we can. There exists a simple OT-based protocol for generating DPF keys in a distributed manner [Ds17].

References

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1292–1303. ACM Press, October 2016.

[Ds17] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 523–535. ACM Press, October / November 2017.