

Private Information Retrieval and Distributed Point Functions

CS 507, Topics in Cryptography: Secure Computation

David Heath

Fall 2026

The techniques we have so far seen in this class are based on XOR/additive secret shares of values. Based on this paradigm, we have constructed secure protocols for Boolean circuits, and now even for RAM programs. Our costs in all such programs run quasi-linearly¹ in the cost of the target program. We've been able to achieve such execution securely in the presence of a semi-honest adversary, and even in the presence of a malicious adversary who corrupts any subset of parties. Again, all of this was essentially built on the notion of secret-shared data.

Today, we will introduce an additional technique for sharing information that is useful in the context of secure computation, and also broadly in cryptographic applications in general. This technique allows parties to secret share *functions* rather than bits, and hence it is appropriately named *function secret sharing* (FSS). The motivation for introducing FSS is that it can improve the efficiency of many cryptographic tasks.

Today we focus on a simple and useful kind of FSS, the *distributed point function* (DPF). We first motivate DPFs through the problem of Private Information Retrieval (PIR), then define DPFs, construct one from any PRG, and finally show how DPFs give GMW an efficient lookup-table gate.

A Motivating Task: Two-Server PIR

Private Information Retrieval is a problem motivated by the desire for a *client* to securely download a piece of information from some database. In particular, suppose there are t non-colluding servers $S_0, \dots, S_{t-1}$, each of whom holds some “database”, which for today will just be a length- n bitstring $x \in \{0, 1\}^n$. A client holds some index α , and she would like to download the database entry x_α while keeping α hidden. Formally, we want semi-honest simulation security against each server. Two notes on this model:

- We do not need to consider subsets of servers, since we assume they do not collude.
- The database x is *not* considered private, so we do not need security against the client. The only goal is to securely deliver x_α .

Moreover, we would like the client to download the database item x_α as efficiently as possible.

Remark 1 (Comparison with ORAM). *PIR and ORAM at the surface appear similar, but at closer inspection the problems are quite different:*

- *ORAM considers an amortized notion of efficiency. Namely, the client submits many read/write queries to x and we only insist that this is efficient in the aggregate. In PIR, each query should be efficient.*
- *In ORAM, there is a single client whose database x is private. In PIR, the database x is public, and it is instructive to think of there being many clients, each of whom wishes to securely access the same x .*
- *Following the above discussion, in PIR the server cost should not scale significantly with the number of clients. Essentially, the server algorithm should be essentially stateless, keeping only the database x in memory. Note that in ORAM this is not the case, as if there are c clients, then the server would have to store separately the encrypted memory for each client.*

¹Here, “quasi” in general hides polynomial factors of the security parameter, polylogarithmic factors of the program cost, and scaling in number of parties.

PIR is challenging. Indeed, suppose there is only one server, i.e. $t = 1$. Here, it is challenging to see how the client can obtain x_α without leaking α , unless² the client simply downloads *everything* from the database. Indeed, this strategy—where the client downloads everything—is called *trivial* PIR, and it is straightforwardly PIR, but inefficient.

However, once more servers are involved, it becomes more plausible that the client might be able to download data more efficiently. Indeed, there exist very beautiful protocols that achieve two-server PIR at sublinear communication cost (e.g. the client uploads/downloads $O(n^{1/3})$ bits), even without using cryptography at all [CGKS95]. Today we will see how one can get an even better two-server PIR protocol, costing only $O(\lambda \log n)$ bits, by using a DPF.

Secret-Shared One Hot Vectors. So, let us restate our setting. We have two servers, S_0 and S_1 , each of whom hold a copy of some length- n public string $x \in \{0, 1\}^n$. The client holds an index $\alpha \in [n]$, and she wishes to obtain x_α without leaking α .

To show a two-server protocol, it will be convenient to introduce the notion of a *one-hot encoding*. A one-hot encoding of α is a string that is all zeros, except at the α -th position, where it is 1.

Definition 1. Let $\alpha \in [n]$ be an index. The one-hot encoding of α is a string $\text{hot}(\alpha) \in \{0, 1\}^n$, defined as follows:

$$\text{hot}(\alpha)_i = \begin{cases} 1 & \text{if } \alpha = i \\ 0 & \text{otherwise} \end{cases}$$

Now, suppose the client constructs an XOR secret share $\llbracket \text{hot}(\alpha) \rrbracket$, and then sends one share to each server. Each server locally computes the *inner product* of this vector with x . That is, the servers jointly compute the following:

$$\begin{aligned} \bigoplus_i (x_i \cdot \llbracket \text{hot}(\alpha) \rrbracket_i) &= \left\llbracket \bigoplus_i x_i \cdot \text{hot}(\alpha)_i \right\rrbracket \\ &= \left\llbracket \bigoplus_i x_i \cdot \begin{cases} 1 & \text{if } \alpha = i \\ 0 & \text{otherwise} \end{cases} \right\rrbracket = \llbracket x_\alpha \rrbracket \end{aligned}$$

From here, the servers can send their two bits to the client, who locally reconstructs x_α .

While this protocol works, so far, it has not improved cost: The total communication cost is still $O(n)$, since the client sends two shares of length n . However, this is still interesting in comparison to the trivial protocol, since the client now *downloads* only $O(1)$ bits.

Remark 2. As an aside in our path to DPFs, the above protocol can be adjusted to achieve $O(\sqrt{n})$ communication. The idea here is a kind of “load balancing” between the cost of uploading data and the cost of downloading data. In particular, we can have the servers split x into $\sqrt{n}$ chunks, each of size $O(\sqrt{n})$. The client now downloads the entire chunk corresponding to their desired bit α . Since the database now only has $\sqrt{n}$ items, the one-hot shares are also of size $\sqrt{n}$. The databases must now send back shares of the entire chosen chunk, but this is also only $O(\sqrt{n})$ bits.

DPFs

One way to understand what a DPF is to reconsider the above protocol. The client wishes to convey to the server secret shares of some long string $\text{hot}(\alpha)$. A key point is this: While the string $\text{hot}(\alpha)$ is long, it has almost no information in it! Indeed, the entire string is described by only a $\log n$ -bit number α . This is somewhat maddening—surely the client doesn’t need to send $O(n)$ bits to convey such a small amount of information!

Indeed, the client does not (assuming that pseudorandom generators exist). A DPF is essentially a way to very significantly compress the secret-sharing of the vector $\text{hot}(\alpha)$. Formally, we can define a DPF as follows:

²In fact, there are ways to achieve efficient single-server PIR from strong encryption schemes, e.g. from fully homomorphic encryption (FHE). We will not look at such schemes today.

Definition 2 (Two Party DPF). A DPF is a pair of algorithms $\text{KeyGen}, \text{Eval}$:

- 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 .
- 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 :

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

In terms of security, we consider a standard simulation-based notion, where each key k_i is indistinguishable from the output of some simulator that gets only the parameters of the DPF (i.e., w and n) [BGI16].

A two-party DPF can be constructed assuming only that PRGs exist, and the resulting scheme will have keys of length only $O(w + \lambda \log n)$ bits.

In other words, the DPF is a cryptographic representation of a secret-shared one-hot vector. In particular, one can interpret DPF keys as a kind of secret sharing of the following *point function*:

$$\text{point}_{\alpha, \beta}(i) = \begin{cases} \beta & \text{if } i = \alpha \\ 0 & \text{otherwise} \end{cases}$$

Note that point functions and one-hot encodings are subtly different. In short, a point function might have an efficient representation, whereas a one-hot encoding is long. Put another way, the scaled one-hot encoding $\beta \cdot \text{hot}(\alpha)$ is equal to the *truth table* for the function $\text{point}_{\alpha, \beta}$.

Given a DPF, here is a much more communication-efficient two-server PIR protocol:

- The client uses $k_0, k_1 \leftarrow \text{KeyGen}(\alpha, 1)$ to share a point function that encodes its desired index α .
- The client sends k_0 to S_0 and k_1 to S_1 .
- The two servers S_b compute the following:

$$\left(\bigoplus_i x_i \cdot \text{Eval}(k_0, i), \bigoplus_i x_i \cdot \text{Eval}(k_1, i) \right) = \llbracket x_\alpha \rrbracket$$

The above equality follows correctness of the DPF and the same reasoning as applied earlier.

- The servers send their shares back to the client, who locally reconstructs the output.

Because efficient DPFs exist, this protocol consumes only $O(\lambda \log n)$ bits of communication!

Exercise 1 (Security of DPF-based PIR). *Prove that the above protocol is secure against each semi-honest server, directly from the DPF security definition. Then show that a party holding both keys recovers α with $O(n)$ PRG evaluations, so the protocol offers no security at all if the servers collude.*

DPFs from PRGs

We now construct such a DPF from any PRG [BGI16], then turn to its use inside MPC. To begin, let us assume that we have access to a PRG G that (slightly more than) doubles the length of its input:

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

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:

$$\begin{aligned} G_0, G_1 : \{0, 1\}^\lambda &\rightarrow \{0, 1\}^{\lambda+1} \\ G_0(x), G_1(x) &= G(x) \end{aligned}$$

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 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:

$$G(\llbracket s \rrbracket) = G((s_0, s_1)) \triangleq (G(s_0), G(s_1))$$

There are two crucial things worth observing about this notation:

- First, suppose that s is a uniform string that is not known to either party. In that case, $G(\llbracket s \rrbracket)$ is a secret sharing of a *pseudorandom* string: server S_0 does not know s_1 , so by PRG security $G(s_1)$, and hence the shared value $G(s_0) \oplus G(s_1)$, is indistinguishable from uniform to S_0 .
- 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 $G(\llbracket 0^\lambda \rrbracket)$ evaluates to $\llbracket 0^{2\lambda+2} \rrbracket$.

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* $\llbracket A \rrbracket$ 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 generates 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, they hold secret shares of an array of the following form:

$$\llbracket 0^\lambda 0, 0^\lambda 0, \dots, 0^\lambda 0, s1, 0^\lambda 0, \dots, 0^\lambda 0, 0^\lambda 0 \rrbracket$$

where $s \in \{0,1\}^\lambda$ is some pseudorandom string. Now, suppose that for each array slot $\llbracket A_i \rrbracket$, the parties apply G_0 and G_1 to the secret-shared “seed part” of that index. There are two cases to consider:

$$\begin{aligned} G_0(\llbracket 0^\lambda \rrbracket) &= \llbracket 0^\lambda 0 \rrbracket & G_1(\llbracket 0^\lambda \rrbracket) &= \llbracket 0^\lambda 0 \rrbracket \\ G_0(\llbracket s \rrbracket) &\stackrel{c}{=} \{\llbracket r \rrbracket \text{ where } r \leftarrow \{0,1\}^{\lambda+1}\} & G_1(\llbracket s \rrbracket) &\stackrel{c}{=} \{\llbracket r \rrbracket \text{ where } r \leftarrow \{0,1\}^{\lambda+1}\} \end{aligned}$$

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 the true seed s was expanded. Specifically, A' looks like this:

$$\llbracket 0^\lambda 0, 0^\lambda 0, 0^\lambda 0, 0^\lambda 0, \dots, 0^\lambda 0, 0^\lambda 0, (R_0, r_0), (R_1, r_1), 0^\lambda 0, 0^\lambda 0, \dots, 0^\lambda 0, 0^\lambda 0, 0^\lambda 0 \rrbracket \quad (1)$$

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 our invariant, the client must somehow “correct” these two positions. To do so, the client will send to the two servers the following *correction word*:

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

Upon receiving correction word $\mathbf{cw} = (R, c_0, c_1)$, the server first expands it into a string $\mathbf{cw}' = (R, c_0, R, c_1)$. 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. (This describes the case where α is even; the odd case is symmetric.)

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

$$\llbracket 0^\lambda 0, 0^\lambda 0, 0^\lambda 0, 0^\lambda 0, \dots, 0^\lambda 0, 0^\lambda 0, (R, c_0), (R, c_1), 0^\lambda 0, 0^\lambda 0, \dots, 0^\lambda 0, 0^\lambda 0, 0^\lambda 0 \rrbracket \quad (2)$$

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 $\mathbf{cw}$ 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, then repeatedly doubles the secret shared array length by including a correction word of length $\lambda + 2$. After $\log n$ doublings, the client is done. Hence, the full key-length is only $O(\lambda \log n)$ bits.

Concretely,

$$k_b = (s_b^{(0)}, t_b^{(0)}, \mathbf{cw}_1, \dots, \mathbf{cw}_{\log n}, \mathbf{cw}_{\text{out}}),$$

where $s_0^{(0)} \oplus s_1^{(0)} = s$ is the root seed and $t_0^{(0)} \oplus t_1^{(0)} = 1$. The correction words are the same in both keys. The two servers' views differ only in $\lambda + 1$ bits, and yet they expand to secret shares of a one-hot vector of length n .

As described, the active leaf holds $(s_\alpha, 1)$ for a pseudorandom s_α , so **Eval** can output the indicator bit, i.e. $\beta = 1$. For general $\beta \in \{0, 1\}^w$, the client appends one final correction word. Let $G' : \{0, 1\}^\lambda \rightarrow \{0, 1\}^w$ be a PRG. Each server expands its leaf seed share with G' ; the shared value at the active leaf is $G'(s_{\alpha,0}) \oplus G'(s_{\alpha,1})$, and the client sends:

$$\mathbf{cw}_{\text{out}} := G'(s_{\alpha,0}) \oplus G'(s_{\alpha,1}) \oplus \beta.$$

Each server XORs $\mathbf{cw}_{\text{out}}$ into its expansion, masked by its indicator-bit share. At the active leaf the shared value becomes β ; everywhere else the seed shares are equal and the indicator shares are equal, so the shared value is 0. This adds w bits to the key, giving $O(\lambda \log n + w)$ in total.

Exercise 2 (Security of the construction). *Prove that the PRG-based DPF satisfies the simulation-based security definition. Proceed by a hybrid argument over levels, maintaining the invariant that, conditioned on everything server S_b holds, server S_{1-b} 's share of the active seed at the current level is uniform.*

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 $O(\log n)$ 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 $O(\log n)$ PRG evaluations.

Exercise 3 (Full-domain evaluation).

1. Show that evaluating a key at all n points costs $O(n)$ PRG evaluations, not $O(n \log n)$.
2. When $w \ll \lambda$, the last levels of the tree are wasteful: a λ -bit leaf seed can be expanded to λ/w outputs at once. Modify the construction to stop the tree at depth $\log(nw/\lambda)$ and show that the key length becomes $O(\lambda \log(nw/\lambda) + \lambda)$.

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 $f : \{0, 1\}^k \rightarrow \{0, 1\}$, and let us assume that k 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 $O(k)$ bits of online communication. To do this, suppose that the parties were given by a dealer a secret-shared one-hot encoding $\llbracket \text{hot}(\alpha) \rrbracket$, as well as $\llbracket \alpha \rrbracket$ for some uniformly random $\alpha \in \{0, 1\}^k$. Now, as GMW runs, they obtain some gate input $\llbracket x \rrbracket$, and they wish to obtain $\llbracket f(x) \rrbracket$.

To achieve this evaluation, the parties first locally compute $\llbracket x \oplus \alpha \rrbracket$, 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 $\llbracket \text{hot}(\alpha) \rrbracket$ by distance $x \oplus \alpha$. Namely, they create a new vector h as follows:

$$\llbracket h_i \rrbracket = \llbracket \text{hot}(\alpha) \rrbracket_{i \oplus x \oplus \alpha}$$

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

$$\bigoplus_i f(i) \cdot \llbracket \text{hot}(x) \rrbracket_i$$

Essentially, this 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 $\llbracket f(x) \rrbracket$. So we are done!

We have successfully computed $\llbracket f(x) \rrbracket$ for *arbitrary* function f , and all that was required was to reveal some difference $x \oplus \alpha$ of length k .

Here is where DPFs can be used. The dealer's one-hot encoding $\llbracket \text{hot}(\alpha) \rrbracket$ can be succinctly transmitted via the point function $\text{point}_{\alpha, 1}$ with a random α : the dealer sends k_b to party b , and each party expands its key by full-domain evaluation. The preprocessing material per gate therefore shrinks from 2^k bits to $O(\lambda k)$ bits. The online cost is unchanged: one round and k bits. This use of FSS as preprocessing for MPC gates is explained in detail in [BGI19].

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.
- [BGI19] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 341–371. Springer, Cham, December 2019.
- [CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *36th FOCS*, pages 41–50. IEEE Computer Society Press, October 1995.
- [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.