# Secret-Shared Shuffle E0312

Anirudh Gupta  Himanshu Kumar

# Overview of 🔐-🤫💨 🔀

TODAY

Problem Statement
Motivation for this method
Dependence Tree
Permute + Share
Share Translation
OPV
Building OPV from OT, PRFs
SSS Protocol full

NOT TODAY

Benes Network
Discussion on Parameters
Optimizations used
CPU Implementation
What can be parallelized and Optimzed
Conclusion

# Problem Statement

Design light(er) weight protocol for computing secret shares of shuffled data i,e. $P_0$. The protocol designed is called “Secret-Shared Shuffle”(SSS)

- 2 Party Protocol
- Semi Honest Adversary

Say database is x and permutation $\pi$, We want at the end, one party should have **r** and other party should get $\boldsymbol{\pi}$**(x)**$\oplus$**r** as shares, where **r** is some pseudorandom vector.

# Permute+Share - Simple but Slow 🫠

1. $P_1$ holds database x and $P_0$ holds the permutation $\pi$
2. $P_1$ uses additive homomorphic encrypts x and sends Enc(x) to $P_0$
3. $P_0$ choose a random r, adds it to get Enc(x⊕r).
4. It then permutes them and sends back to $P_1$ $\pi$(Enc(x⊕r))=Enc($\pi$(x⊕r))
5. $P_1$ can now decrypt them to get $\pi$(x⊕r) and $P_0$'s share is r.

But this is very slow since Public-key operations are extremely slow.

This is 2 rounds, Communication complexity is 2*N*L

# Motivation and Idea

- Public Key based Encryptions are very expensive in cost.

- To avoid such heavy compute intensive approaches, this protocol uses lightweight crypto primitives like XOR’s and PRG’s and does not require public-key operations besides set of OTs

- Various applications of secret shared shuffle:
  - Private set intersection
  - Collaborative filtering
  - Secure computation for RAM programs (Oblivous RAM)

# SSS Protocol 🐐

Paper presents 2 cases -

1) One party has database **x**, other has permutation $\pi$
2) Both parties has their own inputs $\mathbf{x_0}$, $\mathbf{x_1}$ respectively. Nobody owns a permutation.

- For the 1st case, 1 application of Permute+Share is enough.
- For 2nd case, we need to apply 2 rounds of Permute+Share. 🤓

# P+S - New & Shiny 😎

Let $P_1$ own database **x** and $P_0$ own the permutation $\pi$. We want $P_0$ to learn **r** and $P_1$ to learn $\boldsymbol{\pi(x)\oplus r}$.

1. $P_0$ and $P_1$ do something called “Share Translation” , $P_0$ gets $\Delta=\pi(a)\oplus b$ and $P_1$ gets a,b

2. $P_1$ sends $x\oplus a$ to $P_0$ and sets its final share to b

3. $P_0$ sets it’s share to $\pi(x\oplus a)\oplus\Delta = \pi(x)\oplus b$

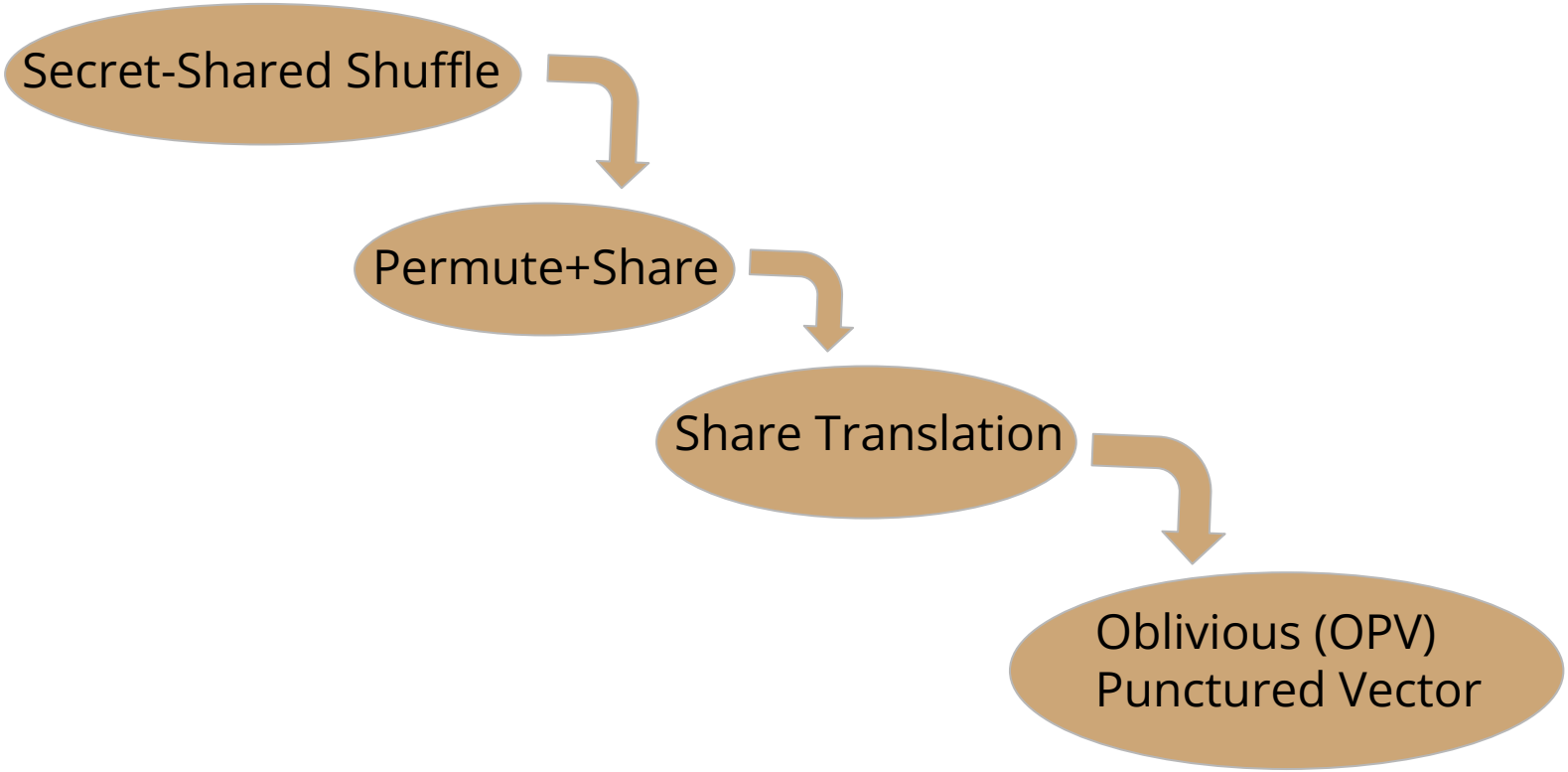
And we are done... but what is Share Translation???

# SSS Protocol for Case 2

P0 and P1 have their input shares of database $x_0$ and $x_1$

0. $P_0$ and $P_1$ each choose a random permutation $\pi_0, \pi_1 \leftarrow S_N$.
1. $P_0$ and $P_1$ run the Permute+Share protocol to apply $\pi_0$ to $\boldsymbol{x}_1$, resulting in shares $\boldsymbol{x}_0^{(1)}$ for $P_0$ and $\boldsymbol{x}_1^{(1)}$ for $P_1$.
2. $P_0$ computes $\boldsymbol{x}_0^{(2)} = \pi_0(\boldsymbol{x}_0) + \boldsymbol{x}_0^{(1)}$.
3. $P_1$ and $P_0$ run the Permute + Share protocol to apply $\pi_1$ to $\boldsymbol{x}_0^{(2)}$, resulting in shares $\boldsymbol{x}_1^{(3)}$ for $P_1$ and $\boldsymbol{x}_0^{(3)}$ for $P_0$.
4. $P_1$ computes $\boldsymbol{x}_1^{(4)} = \pi_1(\boldsymbol{x}_1^{(1)}) + \boldsymbol{x}_1^{(3)}$.
5. $P_0$ outputs $\boldsymbol{x}_0^{(3)}$ and $P_1$ outputs $\boldsymbol{x}_1^{(4)}$.

# Dependence Tree



# Share and Translation(ST)

This new primitive called Share and Translate gives 2 sets of pseudorandom values, one per party, with a special permutation-related dependency between them. They are not chosen by parties.

- $P_1$ receives pseudorandom value **a** and **b**,
- $P_0$ receives $\Delta = \pi(\mathbf{a}) \oplus \mathbf{b}$

**Why not 1 pseudorandom in total? Also if there exists 2, why not that?**

But how is this done??? 🤔

# Oblivious Punctured Vector(OPV)

This is an (n-1)-out-of-n OT, where P0 gives an input $j \in [N]$ and we jointly generate **v** (size N) such that

- P0 learns all the elements of **v** except $j^{th}$, i.e. **v[j]**
- P1 learns the whole vector **v** but doesn't know index j.

**Ques: How do build OPV?**

Answer: Naively, using OT's which is $N^2$ communication

**Ques: How to Use OPV for Share Translation?**

Good Question! 💯 Let's discuss this

# Definitions and Notations

For OPV, Party $P_0$ and $P_1$'s inputs are $((1^\lambda, n), (1^\lambda, n, i)$ and outputs are $(v_0, v_1)$.

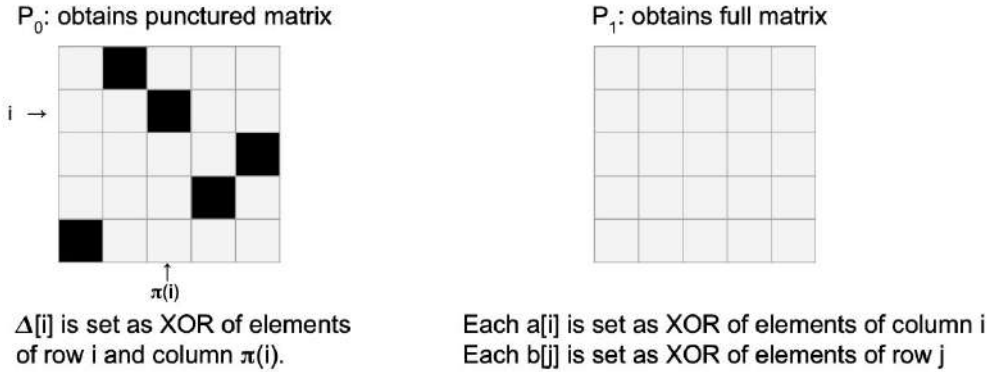
- $\lambda$ is a security parameter(influences the runtime linearly)
- Let **D** be the Field in which they operate. here length of strings in D = $\lambda$
- Since $v_i$ is a vector for $i \in \{0,1\}$, $v_0, v_1 \in [\mathbf{D}]^n$

Hence for any $i \in [n]$ chosen, $v_0[j] = v_1[j] = v[j]$ for all $j \neq i$. For position i, we set $v_0[i]$ = NaN(which is nothing) hence hidden, and $v_1[j]$ is the existing value v[j].

# Building ST from OPV

1. $P_0$ and $P_1$ run N executions of OPV protocol in parallel, where $P_0$ uses $\pi$(i) as its input in execution for all i ∈ [N].

2. We get an NxN matrix $\{v_i[j]\}_{i,j \in N^2}$. Now $P_1$ learns the whole matrix, but the index i for each OPV execution. $P_0$ learns the whole matrix except the $\pi$(i)th position in each row i.e. $v_1[\pi(1)],\ldots,v_N[\pi(N)]$.



# Building ST from OPV(Continued)

3. $P_1$ sets a[i] $\leftarrow v_1[i] \oplus \ldots \oplus v_N[i]$ for all i $\in$ [N] and b[j] $\leftarrow v_j[1] \oplus \ldots \oplus v_j[N]$ for all j $\in$ [N]. This way we obtain the value of a and b.

4. $P_0$ computes $\Delta$[i] by taking sum of column $\pi$[i] except the element it doesnt know which is $v_i[\pi(i)]$.

$$\boldsymbol{\Delta}[i] \leftarrow \left( \bigoplus_{j \neq i} \boldsymbol{v}_j[\pi(i)] \right) \oplus \left( \bigoplus_{j \neq \pi(i)} \boldsymbol{v}_i[j] \right).$$

**CORRECTNESS:** We can also say that $\Delta$[i] = $\sum_{j \neq \pi(i)} \boldsymbol{v}_i[j] - \sum_{j \neq i} \boldsymbol{v}_j[\pi(i)]$

$$= \sum_{j \in [N]} \boldsymbol{v}_i[j] - \sum_{j \in [N]} \boldsymbol{v}_j[\pi(i)] = \boldsymbol{b}_i - \boldsymbol{a}_{\pi(i)}$$

# Building OPV from PRF: Background

- A pseudo-random function (PRF) F is a deterministic algorithm that has two inputs: a key k and an input x

- Defined over (K, X, Y) (finite spaces):
  - the key space K
  - the input space X
  - the output space Y

# Background: PRF Definition

**Experiment b**, for b ∈ {0, 1}:

if $b = 0$: $k \xleftarrow{R} \mathcal{K}$, $f \leftarrow F(k, \cdot)$;
if $b = 1$: $f \xleftarrow{R} \mathrm{Funs}[\mathcal{X}, \mathcal{Y}]$.

k queries $x_1$, $x_2$, … $x_k$

$f(x_1)$, $f(x_2)$, … $f(x_k)$

b’ ∈ {0, 1}

# Background: PRF Definition

**Experiment b**, for b ∈ {0, 1}:

if $b = 0$: $k \xleftarrow{R} \mathcal{K}$, $f \leftarrow F(k, \cdot)$;
if $b = 1$: $f \xleftarrow{R} \mathrm{Funs}[\mathcal{X}, \mathcal{Y}]$.

k queries $x_1$, $x_2$, ... $x_k$

$f(x_1)$, $f(x_2)$, ... $f(x_k)$

b' ∈ {0, 1}

- $W_0$: Event that A outputs 1 in experiment 0
- $W_1$: Event that A outputs 1 in experiment 1
- $|Pr[W_0] - Pr[W_1]| <= negl$

# Background: GGM PRF Tree Construction

- PRG G defined over $(S, S^2)$
  - Seed space S
  - Output space $S^2$
  - $G(s) = (G_0(s), G_1(s))$

- Build PRF F with
  - Key space: S
  - Input space: $\{0, 1\}^l$ (l: polynomial)
  - Output space: S

# Background: GGM PRF Tree Construction

**Algorithm G*(s, x):**

- $s \in S$
- $x \in \{0, 1\}^n$

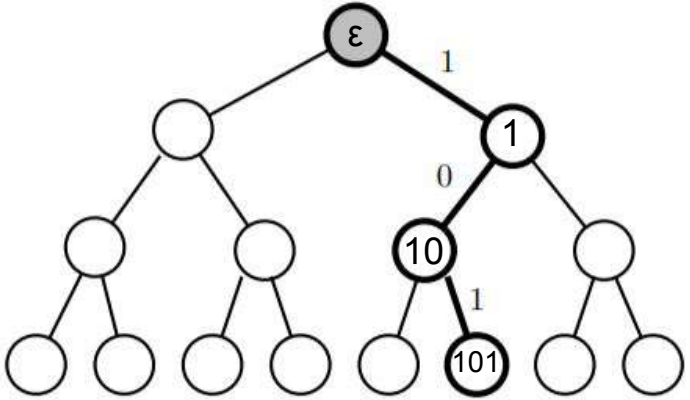
```
t ← s
for i ← 1 to n do:
        b ← x_i
        t ← G_b(t)
output t
```

PRF: F(s, x) = G*(s, x)

# Background: GGM PRF Tree Construction

Visualizing through tree construction:

- Input $x \in \{0, 1\}^l$
- Complete binary tree of height l with $2^l$ leaves
- Bits of input x trace a path from root to a leaf
  - Bit 0: branch left
  - Bit 1: branch right




- Each node is an element of S
- Each node can be uniquely addressed by bit string of length <= l

# Building OPV from PRF

Initially, $P_1$ does the following:

- $P_1$ computes GGM PRF key at random, $seed_\varepsilon$
- Computes vector v:
  - v[i] = PRF($seed_\varepsilon$ , i)
  - Vector v contains values at leaves corresponding to F(1), F(2), … F(N)

- v is $P_1$'s output

# Building OPV from PRF

Let $P_0$'s input index be j.

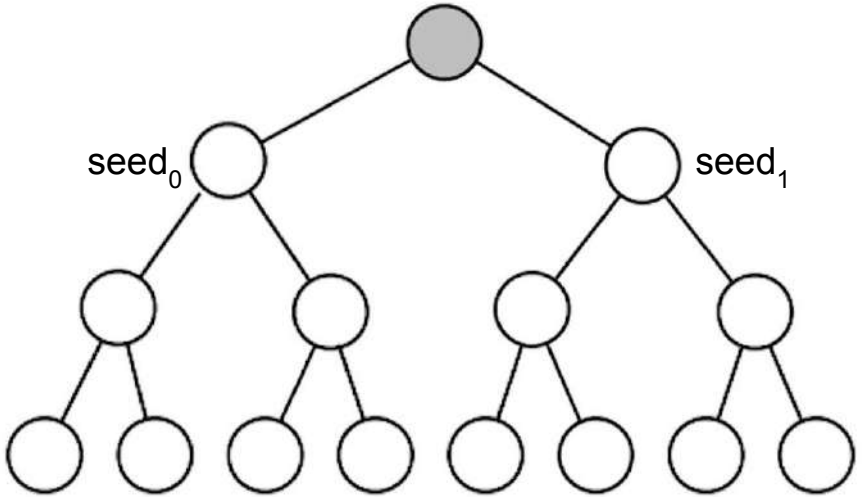
- $P_0$ should learn all leaves F(i) for i ≠ j
- Let's denote internal nodes by $seed_\gamma$
  - γ is binary string corresponding to that node

- For an example, let's assume the first bit of j is 1

# Building OPV from PRF

- For an example, let's assume the first bit of j is 1

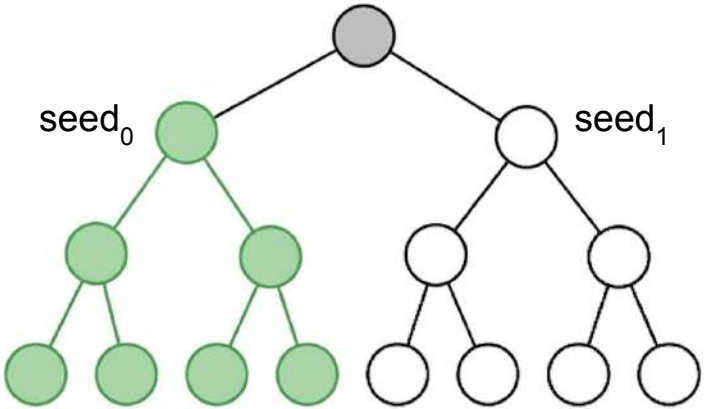
- Parties run 1-out of-2 OT:
  - $P_1$ has $seed_0$ and $seed_1$
  - $P_0$'s bit is complement of first bit of j, i.e., 0



# Building OPV from PRF

- For an example, let's assume the first bit of j is 1

- $P_0$ receives $seed_0$ in OT
- Locally computes entire left subtree
  - F(1), …, F(N/2)

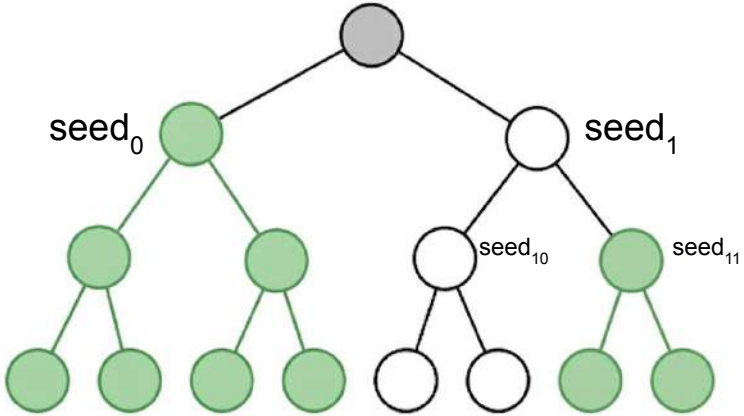


# Building OPV from PRF

- Assume second bit of j is 0
  - j = 10xxxxxxxx

- Parties run 1-out of-4 OT:
  - $P_1$ has $seed_{00}$ , $seed_{01}$ , $seed_{10}$ , $seed_{11}$
  - Flip second bit of j => 1
  - $P_0$ gets $seed_{11}$

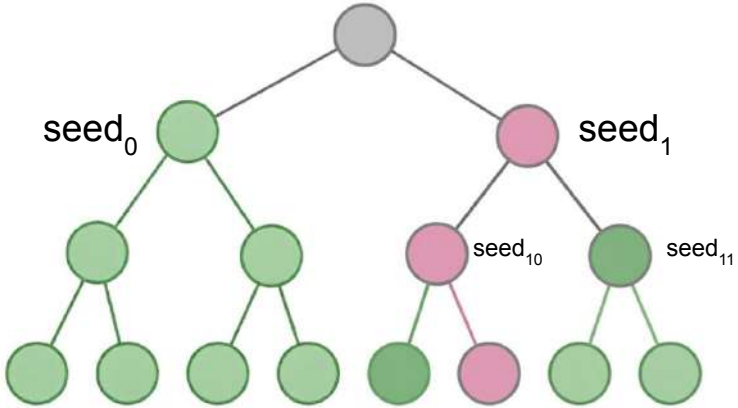
- $P_0$ locally computes F(3N/4)...F(N)



# Building OPV from PRF

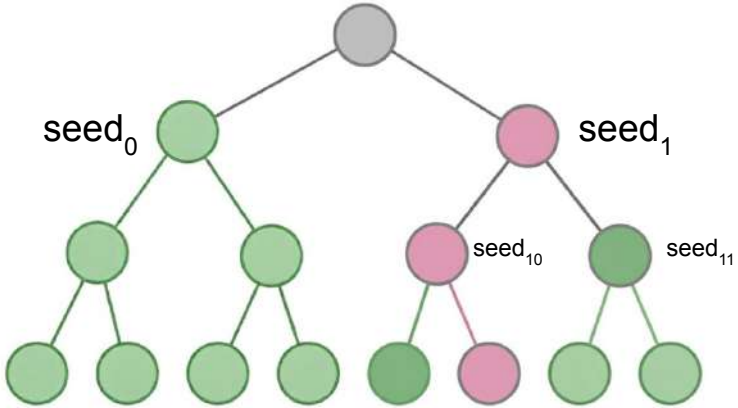
- Assume second bit of j is 0
  - j = 10xxxxxxxx

- 1-out of-8, … 1-out of-N OT
  - $P_0$ knows all nodes and leaves, except nodes corresponding to j
  - $P_0$ constructs vector v using F(1)…F(N) except F(j)



# Building OPV from PRF: Cost Analysis

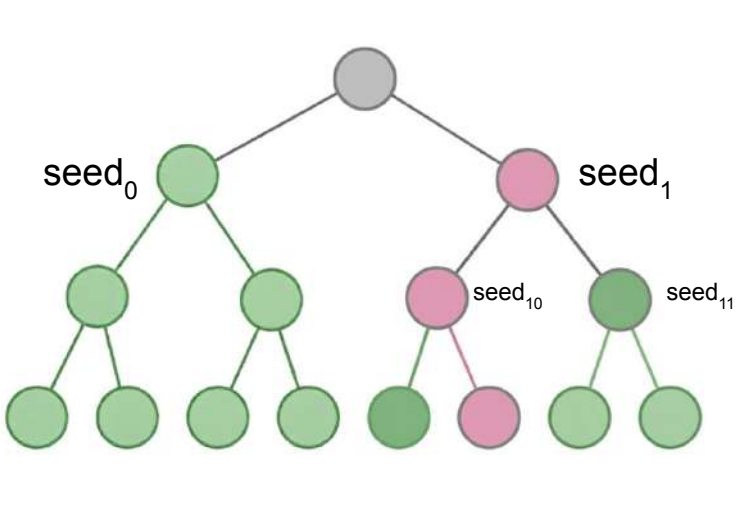
- log N OTs:
  - 1-out of-$2^k$, for levels k



# Building OPV from PRF: Optimization

- log N OTs:
  - 1-out of-$2^k$, for levels k

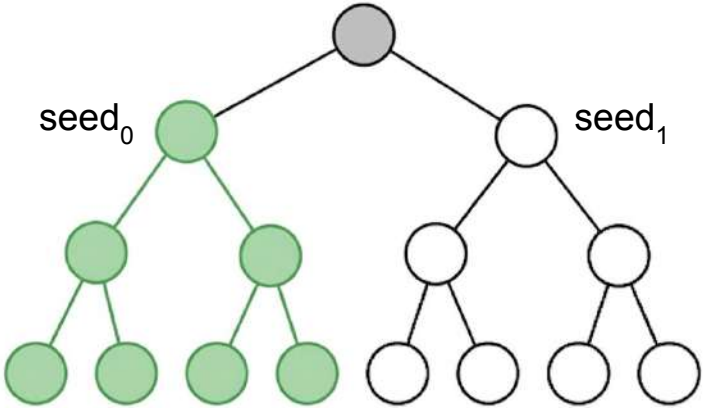
- log N OTs:
  - each 1-out of-2



# Building OPV from PRF: Optimization

- log N OTs:
  - each 1-out of-2

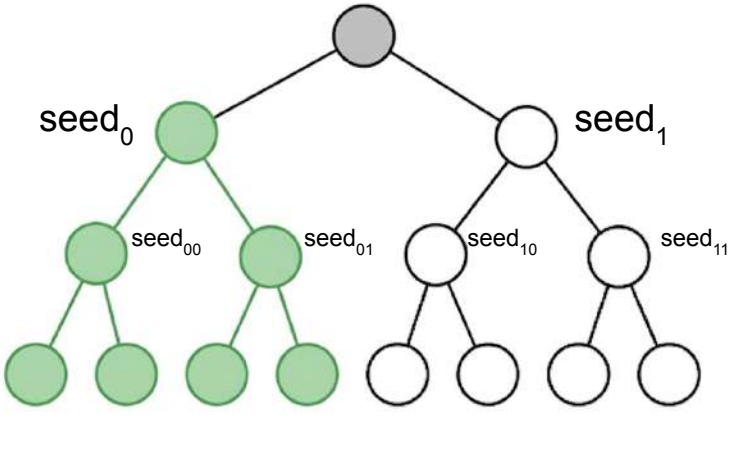
- j = 10xxxxxxxxx
- Assume $P_0$ knows $seed_0$ as before



# Building OPV from PRF: Optimization

- log N OTs:
  - each 1-out of-2

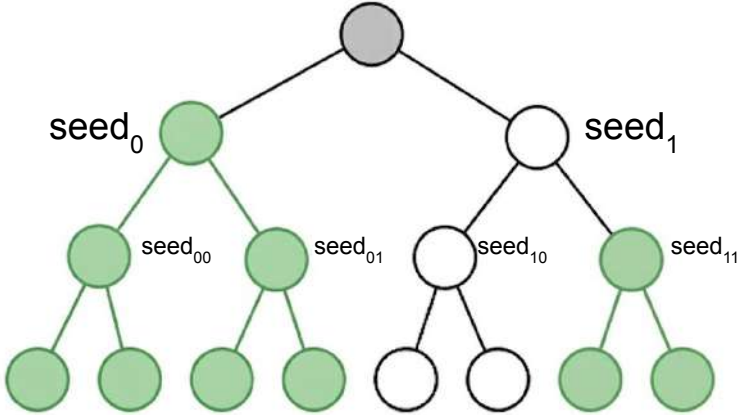
- j = 10xxxxxxxxx
- $seed_{11}$ is learnt as follows:
  - $P_1$ has the following 2 values:
    - $seed_{00} \oplus seed_{10}$
    - $seed_{01} \oplus seed_{11}$



# Building OPV from PRF: Optimization

- log N OTs:
  - each 1-out of-2

- j = 10xxxxxxxxx
- $seed_{11}$ is learnt as follows:
  - $P_1$ has the following 2 values:
    - $seed_{00} \oplus seed_{10}$
    - $seed_{01} \oplus seed_{11}$
  - $P_0$ receives $seed_{01} \oplus seed_{11}$
    - Already knows $seed_{01}$
    - Can compute $seed_{11}$

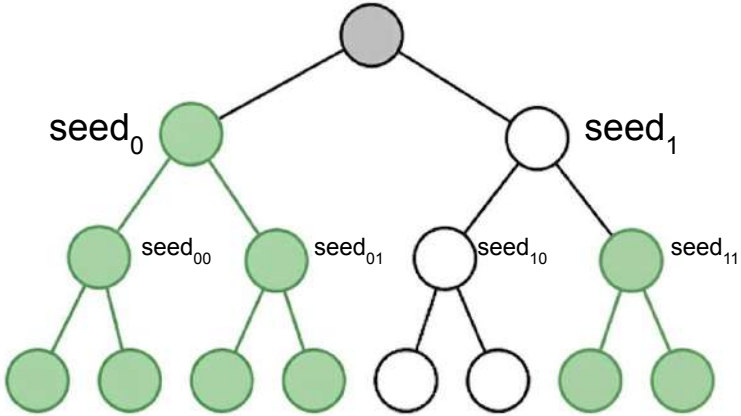


# Building OPV from PRF: Optimization

- log N OTs:
  - each 1-out of-2

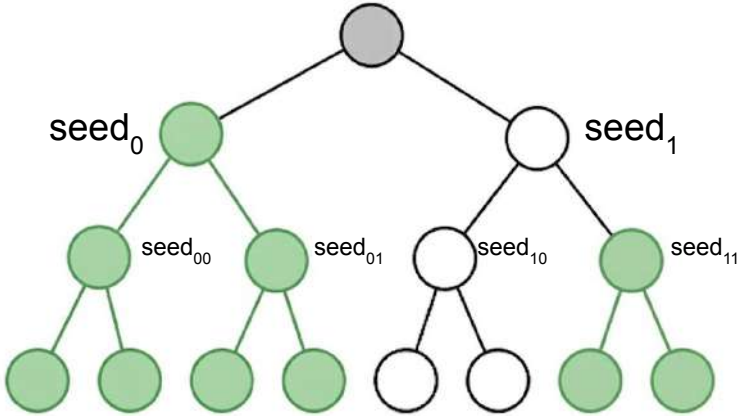
- j = 10xxxxxxxxx
- For each level:

$$M_0 = \bigoplus_{i \in \text{LeftChildren}} seed_i, \quad M_1 = \bigoplus_{i \in \text{RightChildren}} seed_i$$



# Building OPV from PRF: Optimization

- log N OTs:
  - each 1-out of-2




- j = 10xxxxxxxxx
- For each level:

$$M_0 = \bigoplus_{i \in \text{LeftChildren}} seed_i, \quad M_1 = \bigoplus_{i \in \text{RightChildren}} seed_i$$

- Each of $M_0$ and $M_1$ contain exactly one term that $P_0$ doesn't know
  - Xor with other seeds to get that term

# OPV for longer strings?

- We used security parameter λ, but length of string bits ***L*** could be larger.
  - The protocol constructs vector v where each element has length λ
  - We want L >= λ length strings

**What to do?**

1. Run OPV on **D** on previous input to get (v0, v1) as before.

2. Both the parties, uses a PRG **G:** $\{0,1\}^{\lambda}$ -> $\{0,1\}^{l}$ and applied on the output i.e. $v_b' \leftarrow G(v_b)$ for $b \in \{0,1\}$.

**Correctness:** Since we had for **D,** $v_0[j] = v_1[j]$ for all $j \neq i$, therefore we have $v_0'[j] = v_1'[j]$ for all $j \neq i$ by our construction of applying **G.**

# OPV for longer strings?

Advantage of doing this:

- Communication cost for L remains the same as for λ
- Computation cost for PRGs is added

# Communication Complexity - I

| **Primitive** | **Rounds** | **Communication Complexity** | **Why?** |
|---|---|---|---|
| OPV | 2 | λNlogN | N*logN instances of λ-bit 1-out-of-2 OT |
| ST | 2 | Nℓ+λNlogN | OPV cost + 1 Masked Vector (N×ℓ) |
| Permute+Share | 2 | 2Nℓ+λNlogN | 2Nℓ+λlogNST cost + 1 Masked Database (N×ℓ) |

# Number of Rounds

Since we are doing 2 Permute+Share, we would think around 4 rounds, why?

For each Permute+Share = 1 Share Translation + 1 exchange of shares
1 Share Translation = 1 Parallel N * logN executions of OPV
Since we are executing 2 Permute+Shares, hence 2 + 2 = 4

But since OT's are usually complexity intensive, we will end up doing 2*NlogN OT's in the whole go, which is very expensive as N increase.

# 3 rounds 6 messages - Round I

Both parties start by preparing their bits for the permutation they will eventually execute, for OPV GGM Tree.

- **Message 1 (P0→P1):** P0 sends the **OT Queries for π0**.
- **Message 2 (P1→P0):** P1 sends the **OT Queries for π1**.
    - At the end of Round 1, both parties know *what* the OPV Punctured matrix is(for each permutation respectively)

Note: The N*logN will be done in parallel => 1

# 3 rounds 6 messages - Round II

Both parties start by preparing their bits for the permutation they will eventually execute, for OPV GGM Tree.

- **Message 1 (P0→P1):** P0 sends the **OT Queries for π0**.
- **Message 2 (P1→P0):** P1 sends the **OT Queries for π1**.
    - At the end of Round 1, both parties know *what* the OPV Punctured matrix is(for each permutation respectively)

Note: The N*logN will be done in parallel

Now they can locally compute get values of **a,b,$\Delta$** of Share Translation

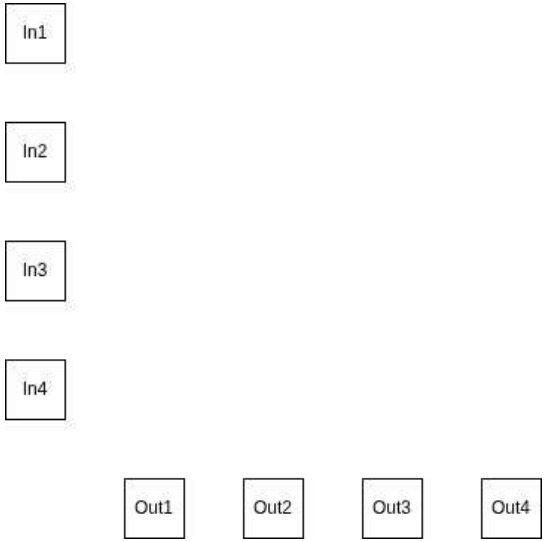
# Optimizing Share Translation

- Share Translation communication complexity is low, but
- Computation complexity $N^2$

- Split permutation π into disjoint permutations of size T
  - using Benes network
  - perform Share Translation independently on T-sized permutations

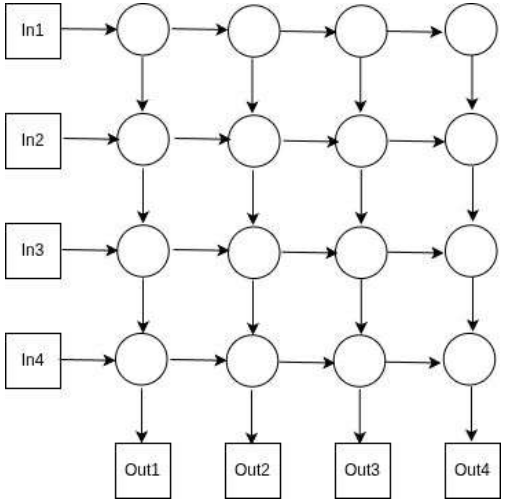
# Benes Network: Introduction

- Comes from the field of communication networks
- Assume NxN network:
  - N machines want to communicate with N other machines



# Benes Network: Introduction

- Switches: Forward packets in the network
- 2x2 switch: 2 inputs, 2 outputs



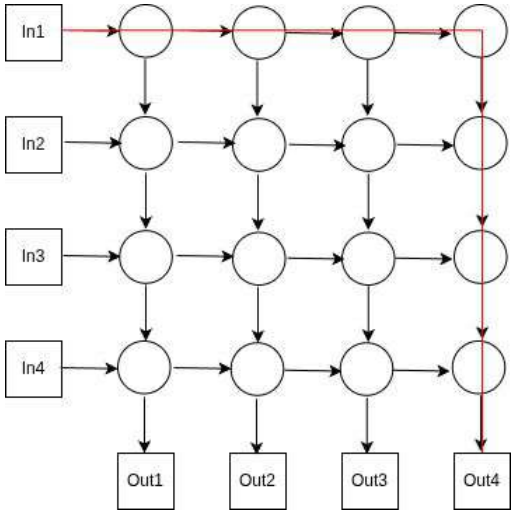
# Permutation Routing Problem

- For input i, route packet to output $\pi(i)$
- $P_{i, \pi(i)}$ : Path from input i to output $\pi(i)$

- Congestion:
  - We have paths $P_{0, \pi(0)}$ , … $P_{n-1, \pi(n-1)}$
  - Congestion: Largest number of paths that pass through a single switch
  - Minimize congestion

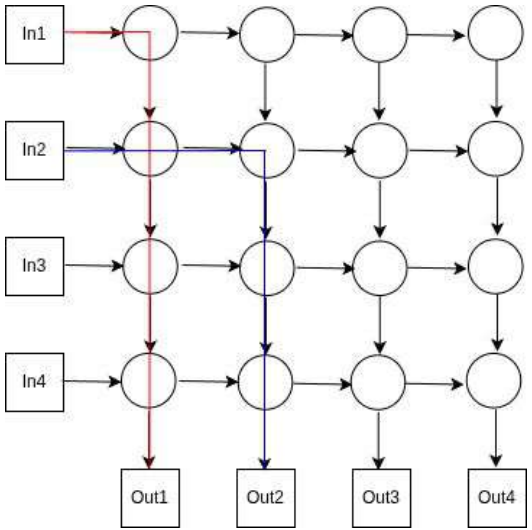
# Permutation Routing Problem

- $\pi(i) = N + 1 - i$
  - reverse permutation
  - Congestion = 1



# Permutation Routing Problem

- π(i) = i
  - identity permutation
  - Congestion = 2

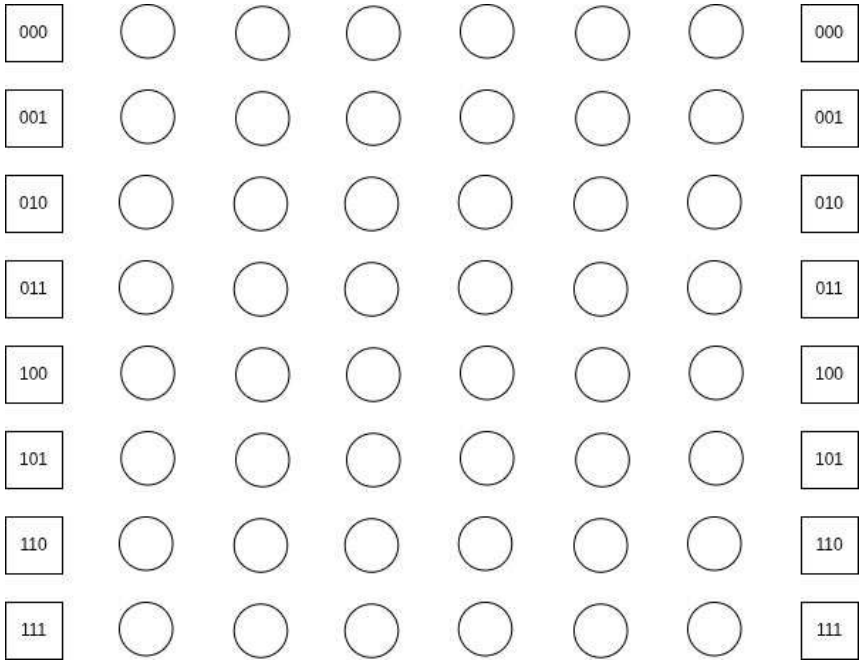


# Benes Network: Construction

- Guarantees congestion = 1 regardless of permutation π
  - We can always construct the paths $P_{i, \pi(i)}$ such that they don't intersect

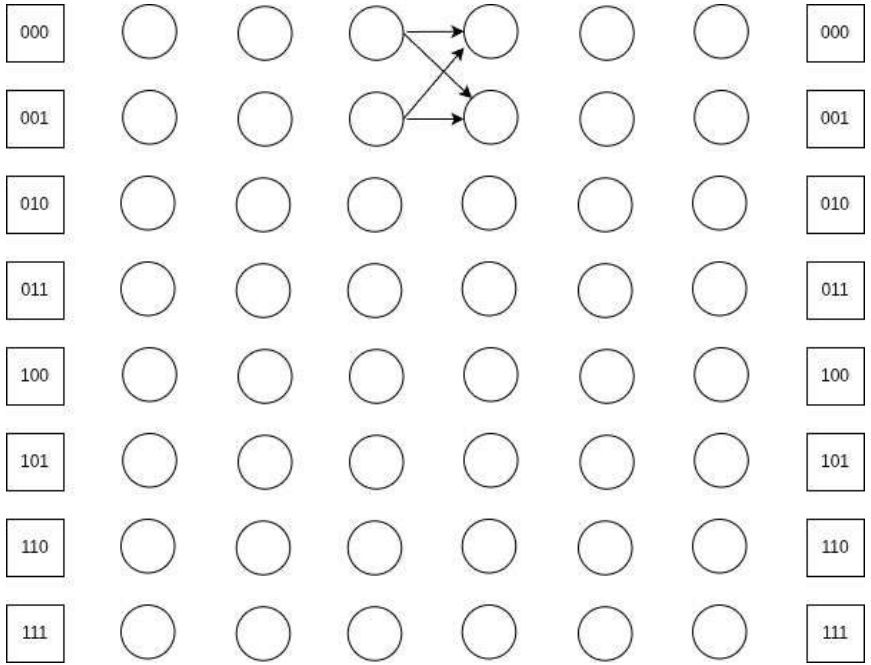
# Benes Network: Construction

- Guarantees congestion = 1 regardless of permutation π
  - We can always construct the paths $P_{i, \pi(i)}$ such that they don't intersect



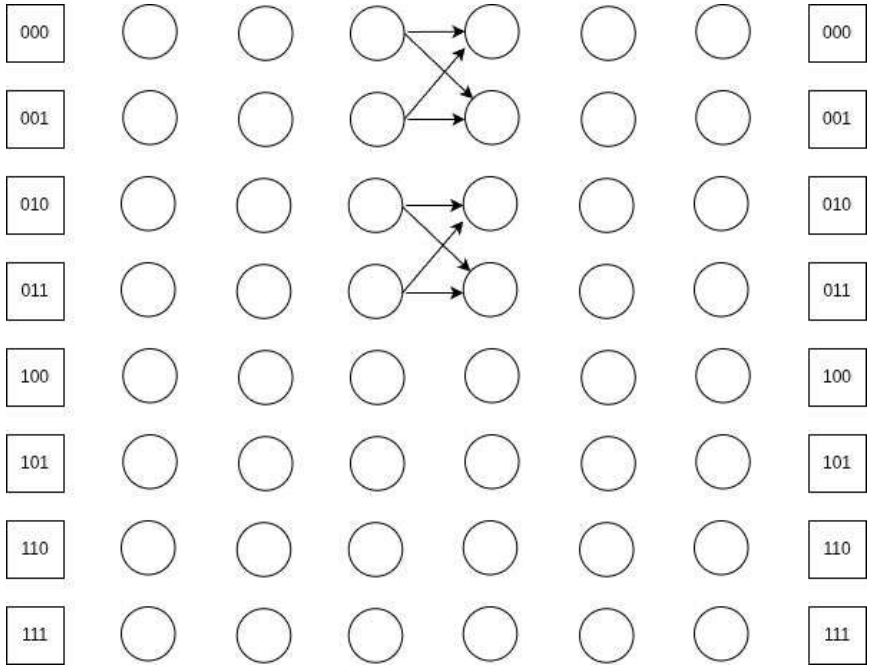
# Benes Network: Construction

- Guarantees congestion = 1 regardless of permutation π
  - We can always construct the paths $P_{i, \pi(i)}$ such that they don't intersect



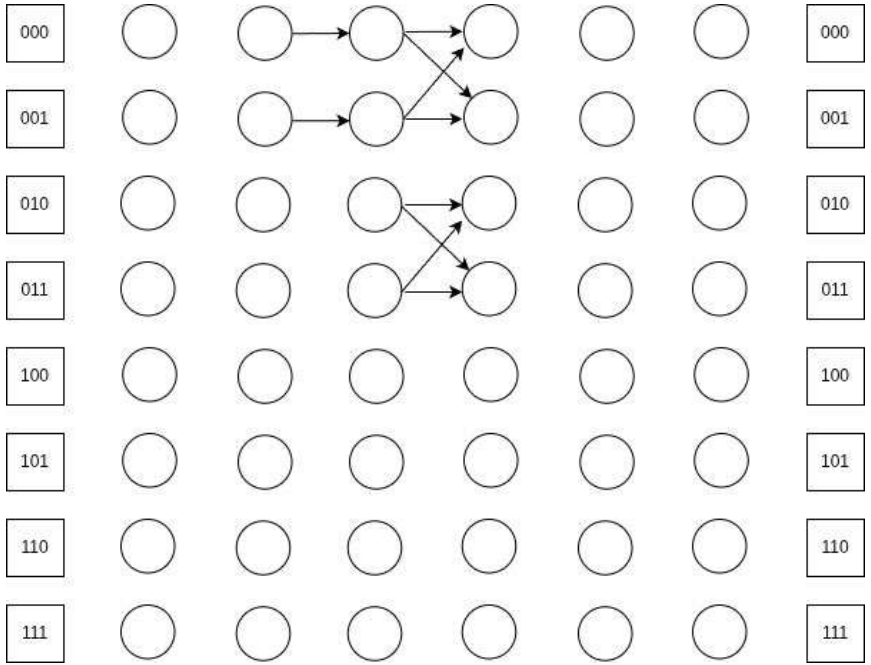
# Benes Network: Construction

- Guarantees congestion = 1 regardless of permutation π
  - We can always construct the paths $P_{i, \pi(i)}$ such that they don't intersect



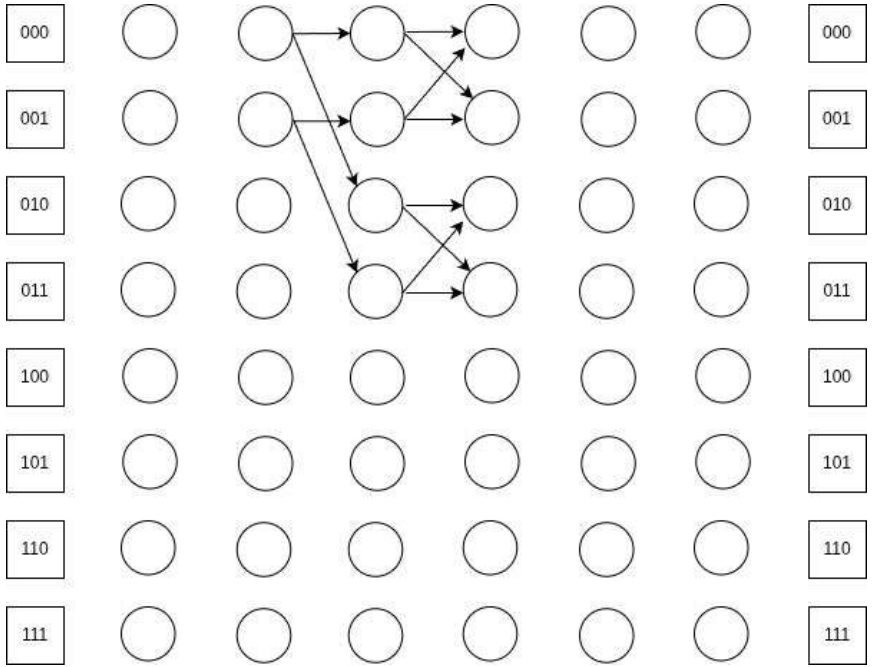
# Benes Network: Construction

- Guarantees congestion = 1 regardless of permutation π
  - We can always construct the paths $P_{i, \pi(i)}$ such that they don't intersect



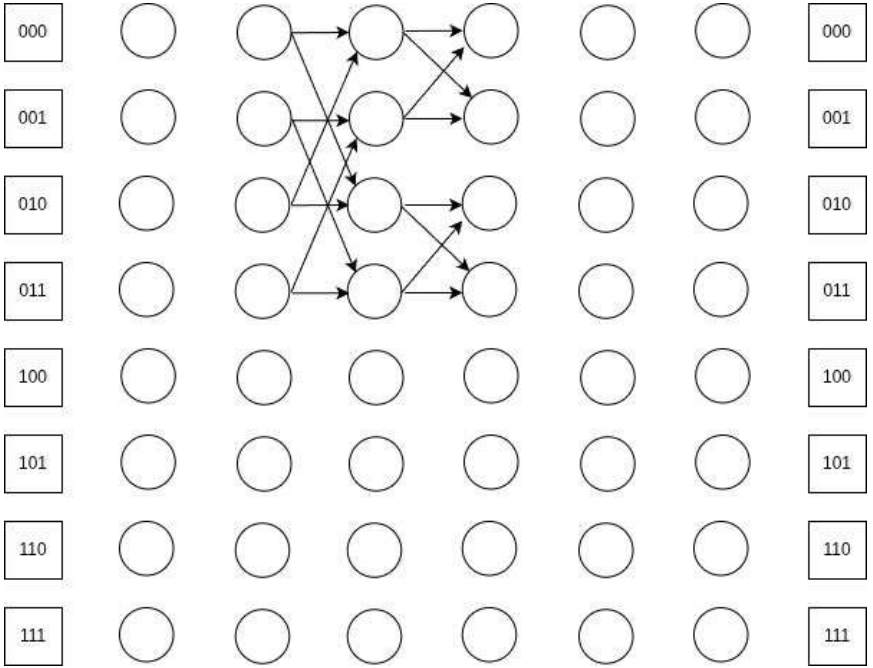
# Benes Network: Construction

- Guarantees congestion = 1 regardless of permutation π
  - We can always construct the paths $P_{i, \pi(i)}$ such that they don't intersect



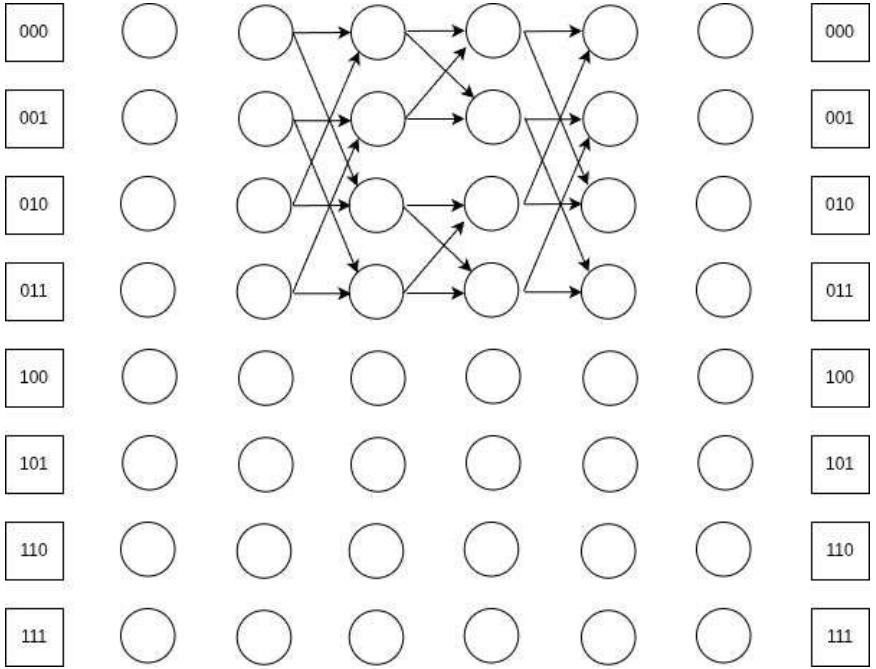
# Benes Network: Construction

- Guarantees congestion = 1 regardless of permutation π
  - We can always construct the paths $P_{i, \pi(i)}$ such that they don't intersect



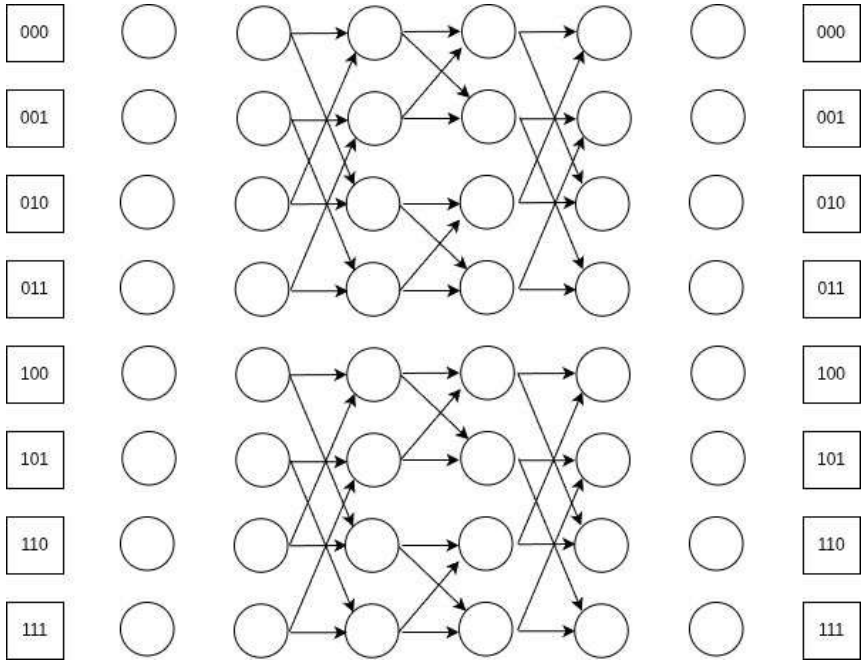
# Benes Network: Construction

- Guarantees congestion = 1 regardless of permutation π
  - We can always construct the paths $P_{i, \pi(i)}$ such that they don't intersect



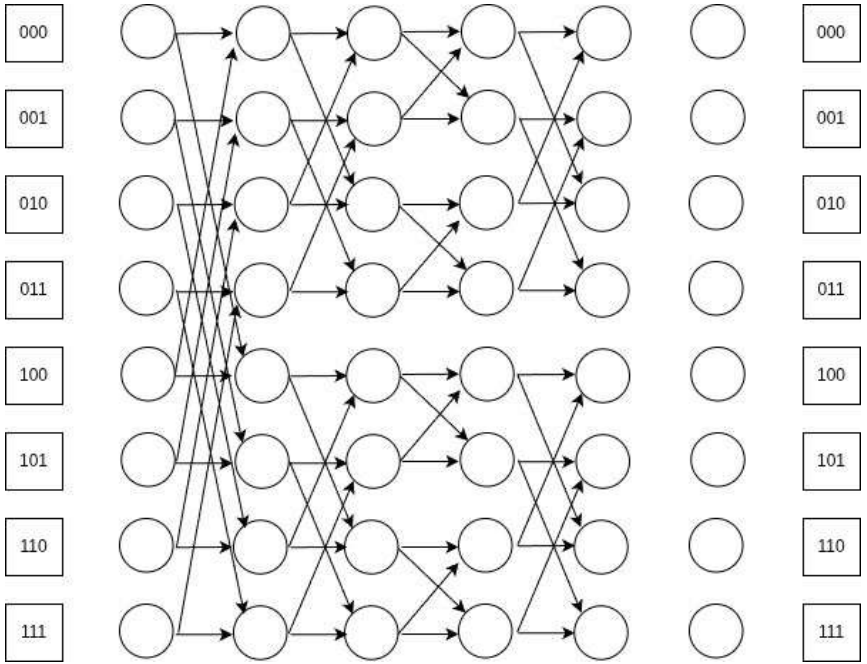
# Benes Network: Construction

- Guarantees congestion = 1 regardless of permutation π
  - We can always construct the paths $P_{i, \pi(i)}$ such that they don't intersect



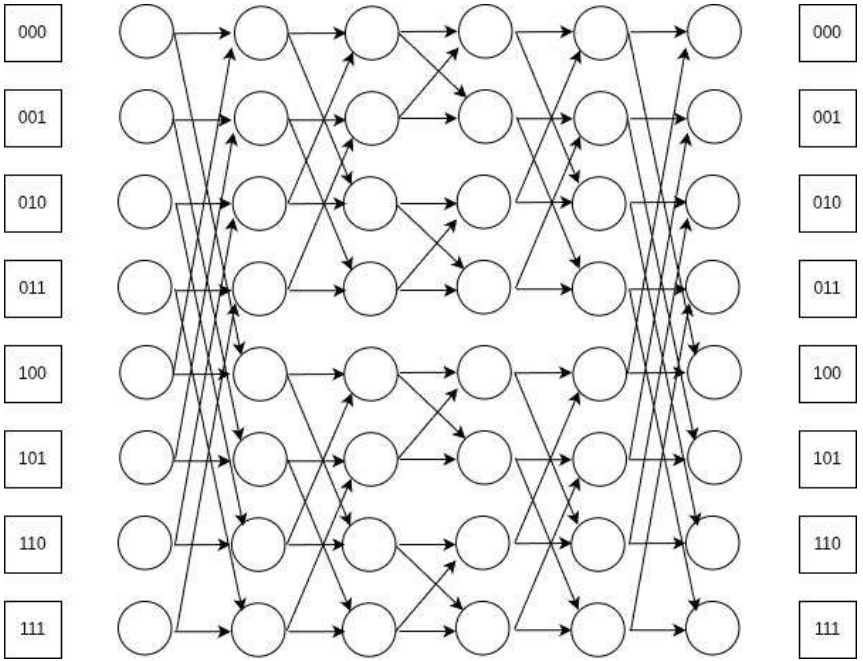
# Benes Network: Construction

- Guarantees congestion = 1 regardless of permutation π
  - We can always construct the paths $P_{i, \pi(i)}$ such that they don't intersect



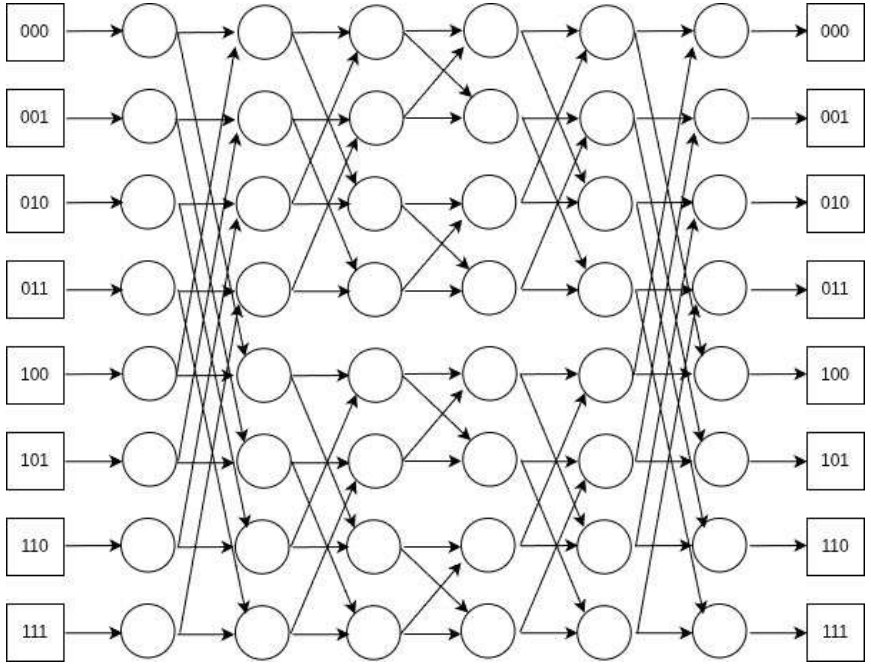
# Benes Network: Construction

- Guarantees congestion = 1 regardless of permutation π
  - We can always construct the paths $P_{i, \pi(i)}$ such that they don't intersect



# Benes Network: Construction

- Guarantees congestion = 1 regardless of permutation π
  - We can always construct the paths $P_{i, \pi(i)}$ such that they don't intersect



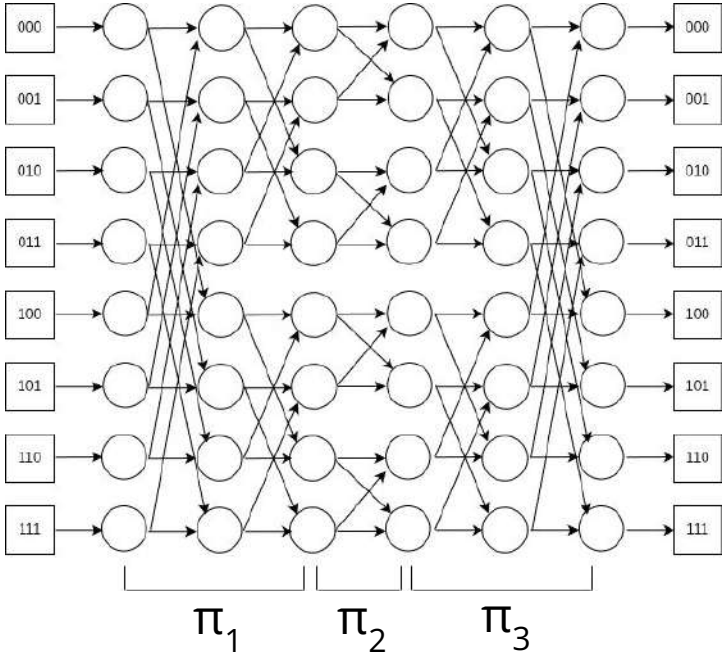
# (T, d)-Subpermutation Representation

- Using Benes network, we want to write $\pi$ as $\pi_1 \circ \pi_2 \ldots \circ \pi_d$
- Each $\pi_i$ is a composition of several disjoint permutations
  - Each $\pi_i$ acts on T elements, for some parameter T

- Relation between T and d:
  - d = 2 ceil(log N/ log T) - 1

- $\pi_i$ acts on log T layers
  - Except for the middle permutation

# (T, d)-Subpermutation Representation



# Implementation on CPU

1. Implementation of ST, Permute+Share and SSS is done on CPU in C++

2. Elaborative test cases with large datasets of upto N = 512 elements with different values of **d** as well.

**Profiling for timings(in ms) (for N = 512)**

**offline=73700.8**
**share_translation=73698.9**
**obliv_setup=73559.6** (Bottleneck)
opv_expand=26.7696
**permute_share=0.492798**

sss_round1=0.24523
sss_round1_local=0.002462
sss_round2=0.249745
sss_round2_local=0.000977
**sss_total=73701.4**

# Implementation Optimizations

- We can optimize the pipeline by running various tasks on GPU like OPV

- Since the bottleneck is Oblivious Transfers used for Share Translation, we also talk about various implementation optimizations useful for increasing speedups of OT’s

# Where can we optimize with GPU

For GPU’s tasks which are data independence among layers, can be easily parallizable and run on GPU’s.

What can be parallelized?
1. **OPV:** All N * logN calls can be run in parallel => parallel construction of punctured matrix P, hence N*logN calls is effectively 1 call.

For local computation in the GGM Tree, children can be computed in parallel for every level one-by-one, so 2^(h-1) => h PRF usages

# Where can we optimize with GPU

2. In **ST** P1 repeatedly reuses the punctured matrix values for computing a and b vectors (xor of rows and cols). So we have two optimizations here:

2.1 Add the punctured matrix in GPU's shared memory, which is a software-managed cache. Advantage, we don't have to worry about whether these accesses will hit in L1 or not, as unlike L1 cache, what's added in shared mem stays in shared memory

2.2 We can implement tiling to compute **a** and **b** vectors, if shared memory can't hold entire punctured matrix. We can divide matrix into some k * k size tiles, and compute partial xors for that part then finally add all xors together

One thread per one row of one tile, we can specify this (same for column)

# Thank You