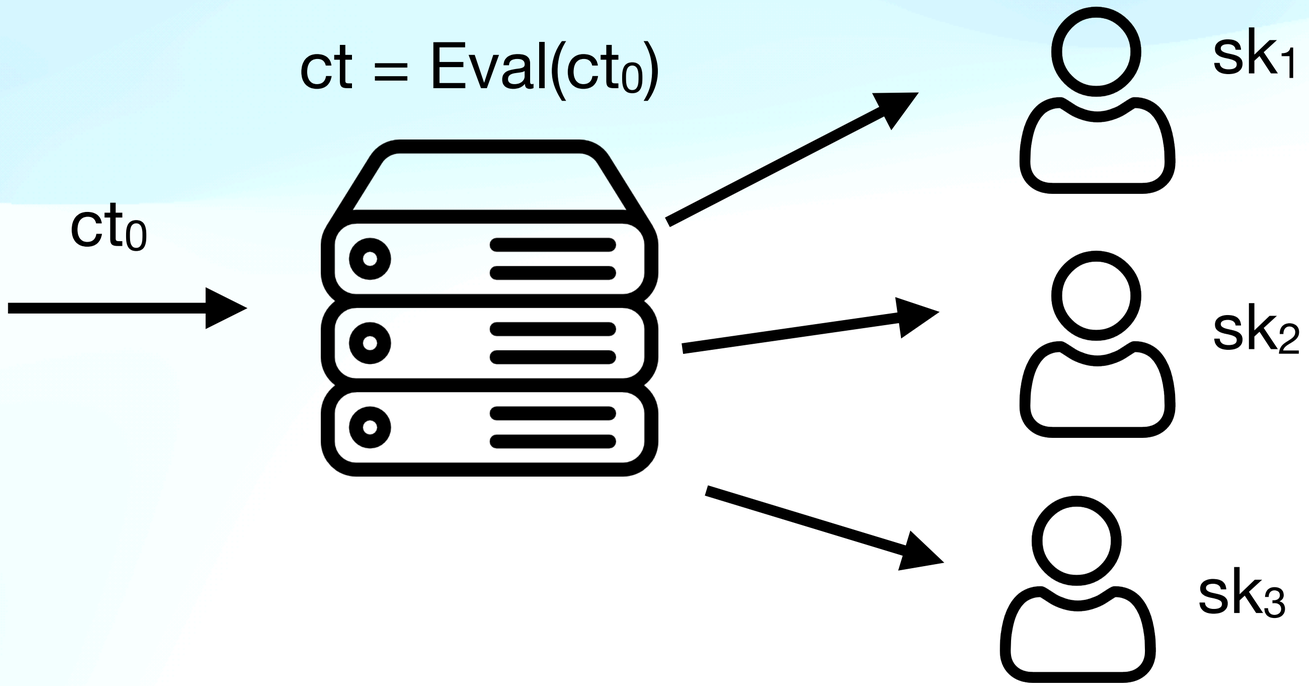
High Throughput Universally Composable Threshold FHE Decryption

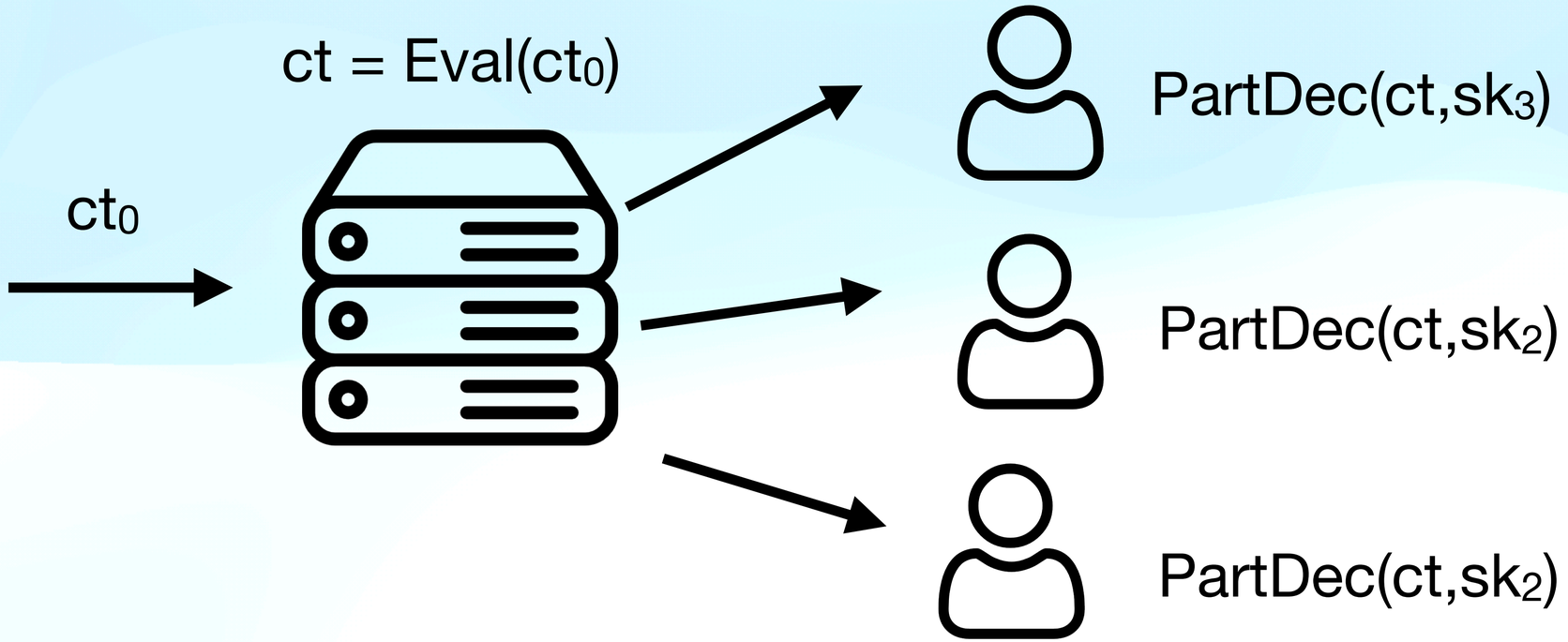
Threshold FHE

- Fully Homomorphic Encryption (FHE) enables arbitrary computation on encrypted data
- Naive application - Trusted Trustee
- Threshold decryption decentralize the decryption capability across multiple parties.
- Allows low-communication multi-party computation (MPC)

Threshold FHE

Preprocess: Secret Share the secret key sk among N parties, $sk_1,...,sk_N$





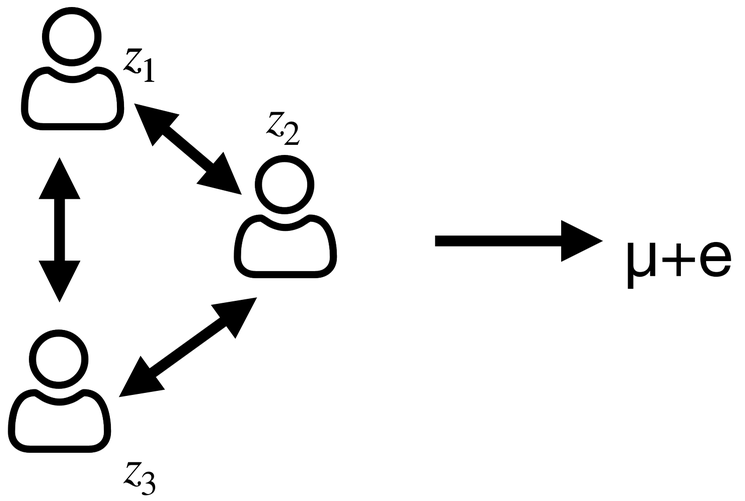
$$d_i \leftarrow C \cdot sk_i$$

LWE encryption scheme

Plaintext form

- **Enc**$_{sk}$(μ)=(a,b)
  - uniform random $a = (a_1,\ldots a_n)$
  - b = ($a \cdot sk$ **+** △μ + e), where e is a small noise, and △ is a scaling factor
- **PartDec**(ct, sk_i): z_i= b - $a \cdot sk_i = (\mu + e)_i$
- = b - $a \cdot$ [sk] = [μ + e] ==> R(∑z_i) = R(μ+e).

Totally insecure!



Noise Flooding

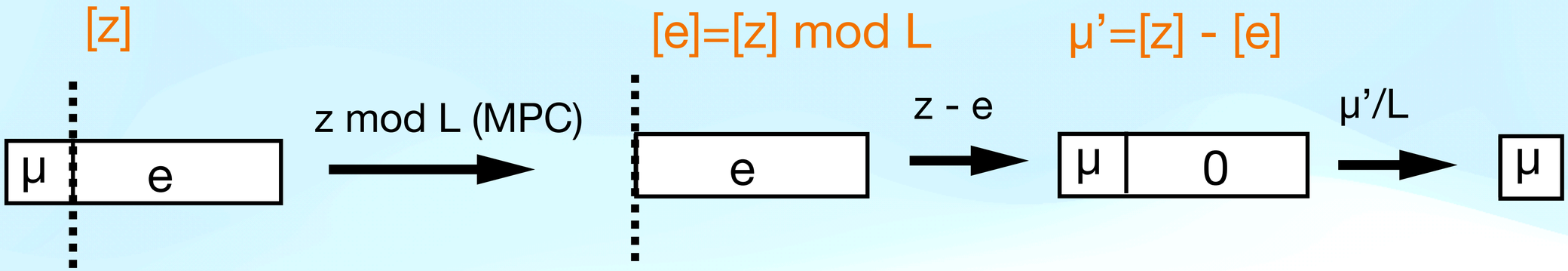
- Adds noise flooding: Adds extra random noise to E ciphertexts before sharing or publishing (Dahl et al. 2023, Boudgoust et al. 2023, Boneh et al. 2018, Brakerski et al. 2025)
  - Parties secret share noise E^{flood}
  - $[d] \leftarrow a \cdot [sk] + [E^{flood}]$
- Requires noise flooding term E (add enough random noise) to statistically mask e (but not too large so it doesn't break correctness).

- Large enough to masks partial decryptions, preventing information leakage
- Small enough to maintains correctness after combination

Noise Flooding Comes at a High Cost

- Large noise term => large modulus => large dimension => slower homomorphic operations
- Prior works on simulation security relied on heavy ZKP, or inefficient secret sharing schemes, leaked security model, added assumptions, weaker security(game based security)
- Dahl et al. (2023) use relatively small modulus (via “switch-n squash” procedure) for increase the cipher text size. But this requires heavy bootstrapping just before threshold decryption.

Our Decryption Protocol



- Π_{Decrypt}:
  - Each party $[z] \leftarrow a \cdot [sk]$
    - Jointly compute $[e] \leftarrow [z] \bmod L$ (via call to Mod-L([z],[r]), from preprocessed sharing of r)
  - Open $\mu' \leftarrow [z] - [e]$
  - Output μ'/L

Mod-L : Deterministic MPC Rounding

- Main challenge: compute [e] = [z mod L]

Mod-L([z],[r]):

  - [z'] = [z] + [r]
  - z' <- Open($[z']_L$) (reconstructs z' = z + r Mod L)
  - e = [z'] - [r]
- But e should reflect [z] - [r] mod . Two cases z' - r < L then e = z' - r Mod L. However, since z
- However since z$\in$ [0,L], r$\in$ [0,L] if z' - r < 0 we to add a correction (adding L)
- Namely the parties jointly compute u=(z'<[r])

Mod-L : Deterministic MPC Rounding

- Mod-L receives sharing of z (from $\prod_{Decrypt}$) and random sharing [r] from preprocessing.

Input([z],[r]):

- [z'] = [z] + [r]
- z' <- Open($[z']_L$) (open only the last L bits)
- e = z' - [r] + L·[u] (mod L)
  - where [u] = (z' <? r) ∈ {0,1} (through LTZ(z',[r]))

Case 1: $z' \geq r$, u=0

[e] = [z'] - [r] = [z] mod L

Case 2: $z' < r$, u=1

[e] = [z'] - [r] ∈ [-L,0]

Procedures LTRand and Mod-L

- LTRand API
  - **Input**: z (mod K)
  - **Output**:
    - z’ = z + r mod L
    - [r] is a stored value
    - u = (z’<r) = (z’-r<0) \in {0,1}
- Mod-L([z])
    - Call (z’,[r],[u]) <- LTRand([z])
    - [e] <- z’ - [r] + [u] * L

Basic LTRand

- Offline - online structure
- Parties compute a random shares [r] for $r \in L$ (offline)
- Compute the lookup table for the function LTZ(z'-r) for all z'.
- $LTZ(z'[i] - r) \in \{0,1\}$

z'	0	1	2	3	4	5	6	7
LTZ(z'-r), r=3	1	1	1	0	0	0	0	0

Naive LTRand procedure

- Lookup table size is $L = 2^l$ of the function LTZ(z'-r).

Lookup at z'=2

z'	0	1	2	3	4	5	6	7
LTZ(z'-r), r=3	1	1	1	0	0	0	0	0

MPC of LTRand procedure

- Prepare offline the table LTZ(z'-r).

z'	0	1	2	3	4	5	6	7
LTZ(z'-r), r=3	1	1	1	0	0	0	0	0
Party 1	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
Party 2	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
Party 3	$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$

- For additive shares, $a_{1,i} + a_{2,i} + a_{3,i} = LTZ(z'[i]-r) \in \{0,1\}$

Problem - LTZ Table is too large

- For 64 bit L, the table is of size 2^{64}
- LTZ is broken to
  - d tables of the Sign function, each of size B=L/d
  - one ModLTZ over the sign tables
- $$Sign(x-r) = Sign\left(\sum_{i\in[d]}(x_i - r_i)\cdot B^i\right) = Sign(\sum_{i\in[d]} Sign(x_i - r_i)\cdot 2^i)$$
- $$(x \overset{?}{<} r) = (x - r \overset{?}{<} 0) = LTZ(\sum_{i\in[d]} Sign(x_i - r_i)\cdot 2^i)$$
- In the paper we use ModLTZ which represent the function LTZ mod 2D

Breaking LTZ to Sign and LTZ tables

1

x	0	1	2	3
Sign(x_1-r_1)	-1	-1	0	1

2

x	0	1	2	3
Sign(x_2-r_2)	-1	0	1	1

.
.
.

d

x	0	1	2	3
Sign(x_n-r_n)	-1	-1	-1	0

$$D = 2^d$$

z	-D	-D+1	…	D
LTZ(z-s)				

$$LTZ(\sum_{i \in [d]} Sign(x_i - r_i) \cdot 2^i)$$

Offline Phase

- Prepare lookup tables for Sign and LTZ gates before the ciphertext is received
  1. Compute [r] via repetive calls to Fabb on RandBit
  2. Prepares one a vector p of the shares of all subset product of the bits of r (based on 1)
  3. Based on 2, construction of the gates are linear combination of the elements of p

Efficiency of The protocol

- Three **rounds** (in the basic implementation).
  - First two round required for the reveal of the masked error (for Sign and LTZ).
  - Third round each party sends its clean shared.
- Total **communication**: l + (d+1) + k where:
  - First round: l-bits (masking l last bits of z)
  - Second d+1 bits (masking d+1 value of LTZ)
  - Third round k bits (shares of the plaintext)
- Total **storage**: d Sign tables of size $2^b \cdot (d+1)$ and one LTZ table of size $2^{d+1} \cdot m$
- Typical choice of parameters: k=64, b=8, d=7, m=1

Security of Π_{decrypt}

- We use ideal functionality Fabb to abstract away all the MPC operations (addition, multiplication, bit generation, open shares)
- We prove UC in the Fabb hybrid model
- The abstraction allows to implement the scheme in various adversarial models:
  - Dishonest majority (e.g., via SPDZ2^k)
  - Honest majority with guaranteed delivery (e.g., SSS with error correction)
  - Adaptive adversary
  - etc.

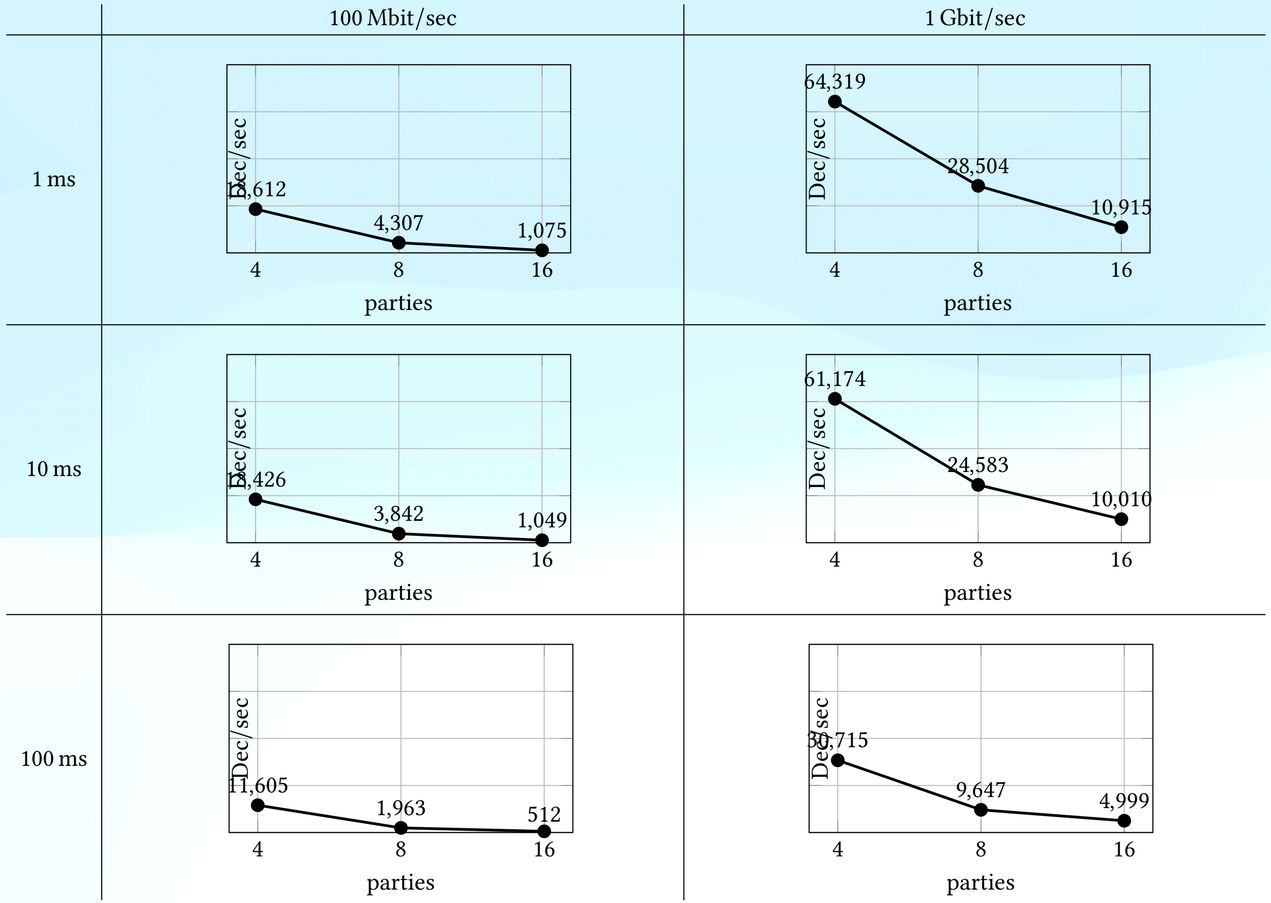
Evaluation of the Online phase

- Implemented under dishonest majority setting (Fabb)
- Measure time (in ms) of the online decryption protocol, for a single ciphertex

	4 **Parties**	8 **Parties**	16 **Parties**
Ping Time 1 ms	8.483	10.086	24.230
Ping Time 10 ms	55.616	55.490	56.979

- LWE parameters: $(n, q, p) = (1024, 2^{64}, 2)$
- For the sign gate we used d=8 blocks of size 8 bits
- Even in 10 ms adding parties there is almost no affect of the number of parties on increased latency.

Throughput under varying network conditions



- Network is a bottleneck for small latencies, but as we increase to higher ping time, the network is the bottleneck.
- Scale to thousands and even tens of thousands of decryptions per second on a single server

Evaluation of the Online phase

Latency and throughput for 4 parties with 1 ms ping time and 1 Gbit bandwidth

Protocol	Latency (ms)	Throughput (Dec/sec)
Noah's ark	315.62	3.18
Our protocol	8.48	64319

Achieve ~37x improvement in latency compared to SOTA!

Achieve ~20000x improvement in throughput compared to SOTA!