

Collaborative attacks extend

Zizheng Liu, Bharat Bhargava

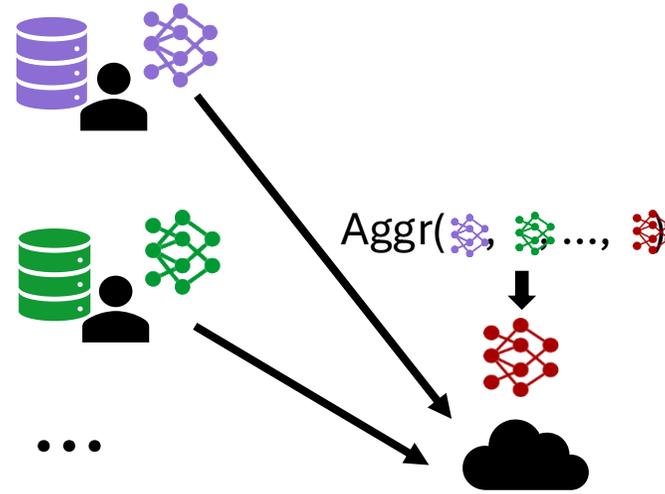
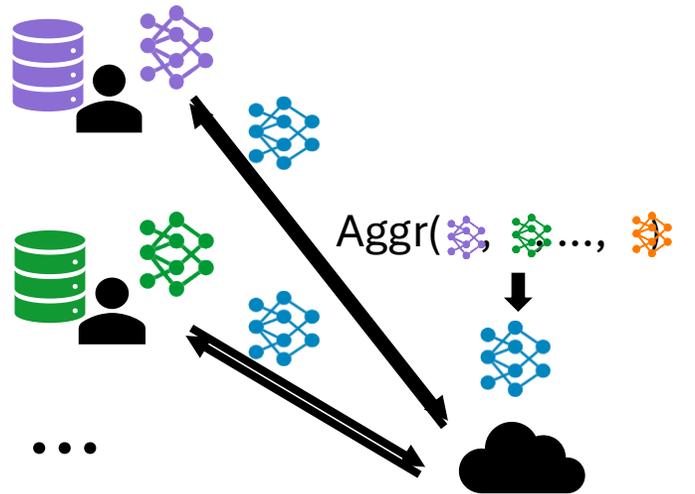
Robust Secure Aggregation for DFL

FL Security and Privacy

Federated Learning (FL)

Security: Poisoning attack

Privacy leakage



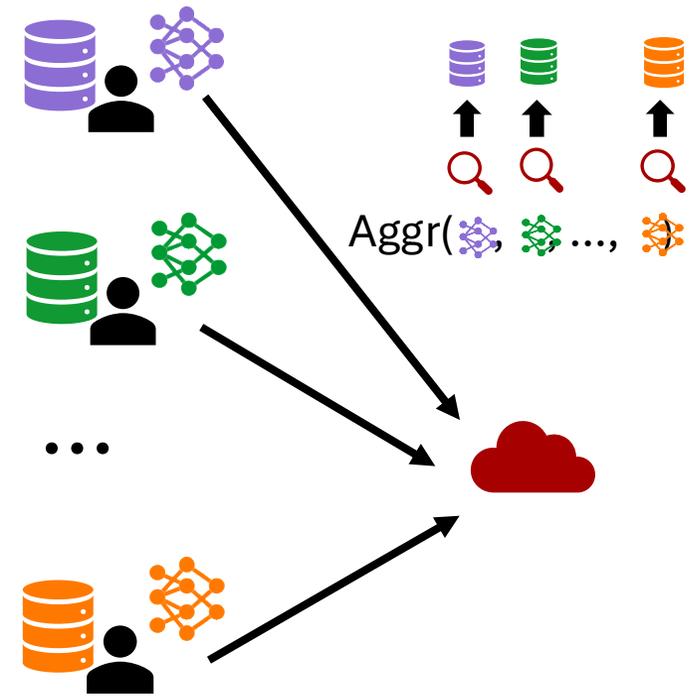
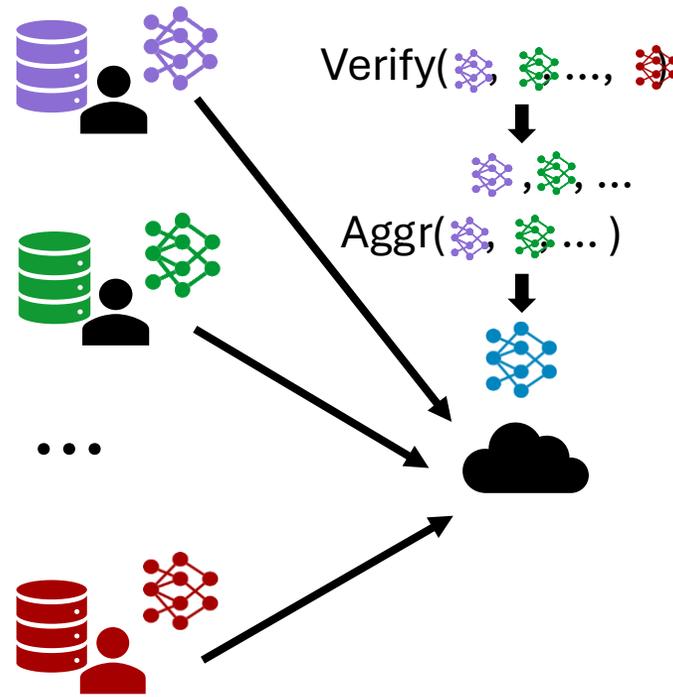
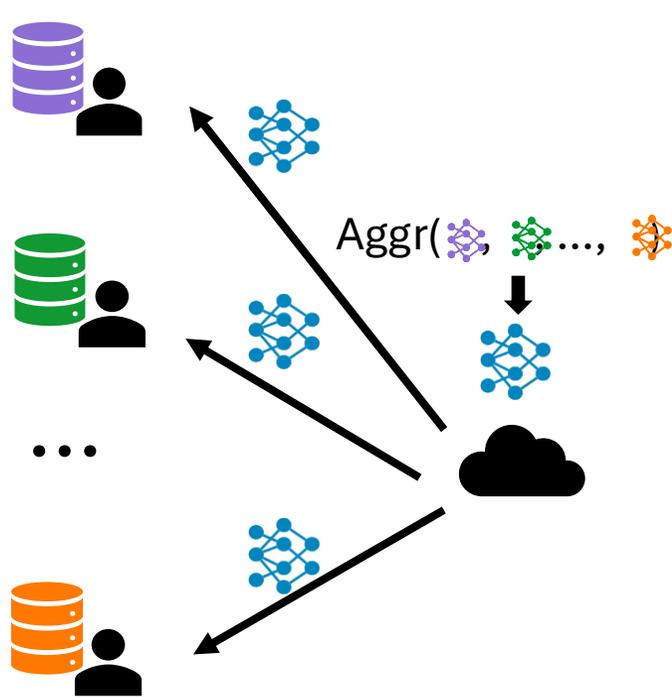
Robust Secure Aggregation for DFL

FL Security and Privacy

Federated Learning (FL)

Security: Poisoning attack

Privacy leakage

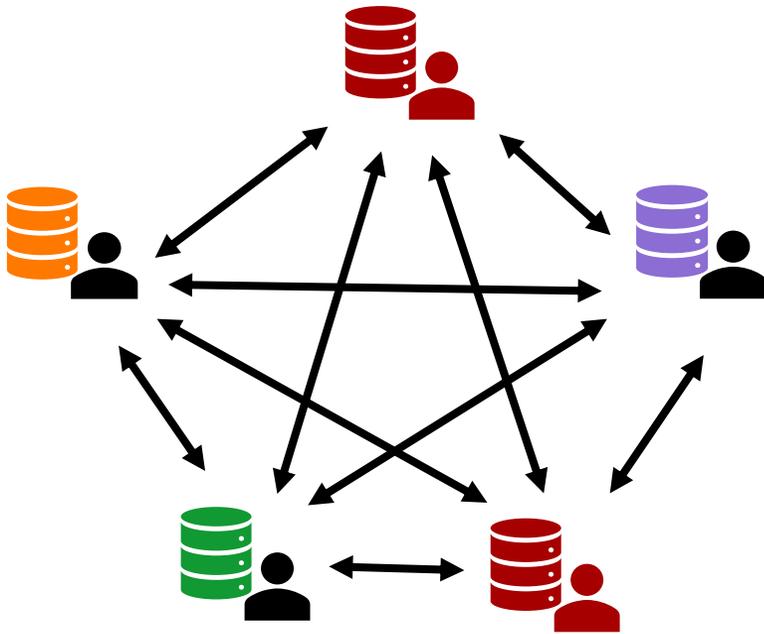


$$Verify(W_1, W_2, \dots, W_n) = W_1, W_2, \dots$$

Robust Secure Aggregation for DFL

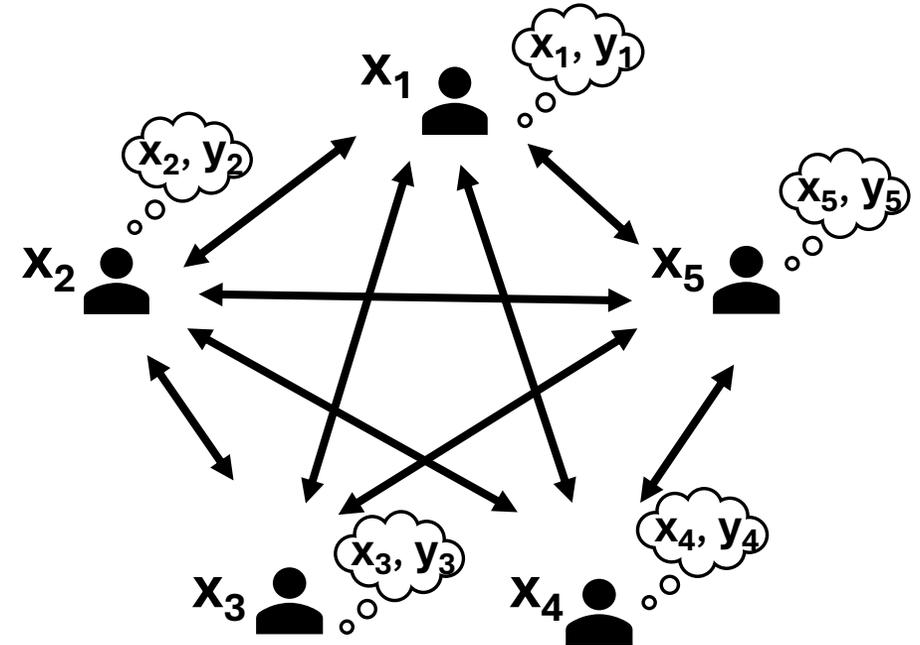
Decentralized FL and Secure Multi-party Computation

Decentralized Federated Learning MPC: Secure Multi-Party Computation



$$\text{Verify}(W_1, W_2, \dots, W_n) = W_1, W_2, \dots$$

$$\text{Aggr}(W_1, W_2, \dots, W_n) = G$$



$$f(x_1, x_2, x_3, x_4, x_5) = y_1, y_2, y_3, y_4, y_5$$

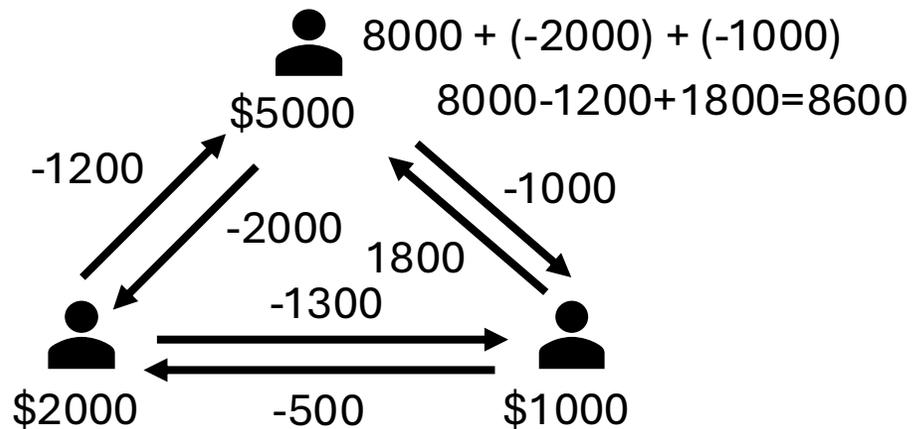
Secure Aggregation: $f() := \text{Aggr}()$; $x_i := W_i$; $y_1 = \dots = y_n := G$

Robust Secure Aggregation: $f() := \text{Aggr}(\text{Verify}())$; $x_i := W_i$; $y_1 = \dots = y_n := G$

Robust and Secure Aggregation for DFL

An example of MPC

MPC Example: compute average salary



$$\begin{aligned}
 &8000 + (-2000) + (-1000) \\
 &8000 - 1200 + 1800 = 8600 \\
 &8600 + 2000 - 2600 = 8000
 \end{aligned}$$

Additive secret sharing: needs all secret shares to reconstruct the secret

- **If all participants are honest**, then by replacing salaries with model parameters and setting the aggregation function $\text{Aggr}()$ to $\text{Avg}()$, the problem is resolved.
- If some participants submit malicious inputs, a $\text{Verify}()$ function must be introduced.
 - **Verify() involves more than simple addition; it requires additional computational operations.**
- **Beyond injecting malicious inputs, an adversary may also cheat during the secure computation process.**
 - For example, the adversary may perform incorrect local computations to prevent the protocol from producing an output.
 - Alternatively, the adversary may simply refuse to participate in the computation.

Robust and Secure Aggregation for DFL

Threat Model

From Machine Learning Perspective

Security: Use **Verify()** to filter out malicious models.

Privacy: Use secure multiparty computation to aggregate models (execute **Aggr()**).

Security and Privacy:

- Use secure multiparty computation to execute **Verify()**, then perform **Aggr()** on the accepted models.

From MPC Perspective

Execute more operations

- $\text{Verify}(W_t, W_r) = \mathbb{I} \|W_t - W_r\|_2 \leq \rho$
 - $\|\cdot\|_2$ needs multiplication and square root computation

Threat Models

- Semi-honest: all parties behave honestly to follow the protocol
- Malicious: parties deviate
 - Secure with abort: halt the protocol when cheating detected
 - **Guaranteed output delivery: find the deviating party, exclude it and continue**

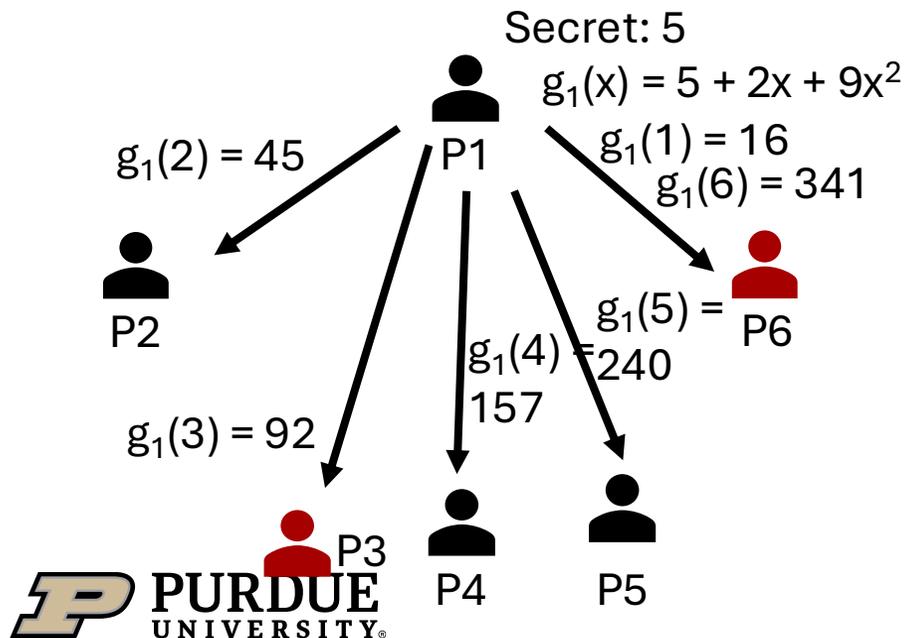
Robust and Secure Aggregation for DFL

Shamir's Secret Sharing and DN multiplication Protocol

Additive secret sharing: needs all secret shares to reconstruct the secret

Shamir's n - t secret sharing **Any $t+1$** parties can reconstruct the secret

$n = 6$ parties, $t = 2$ are corrupted, honest majority, $2t + 1 \leq n$



Shamir SS Properties

- Denote $[x]_t$ as a degree- t sharing of x . $[(1,16), \dots, (6,341)]$ is a degree-2 sharing of 5.
 - $[x]_t + [y]_t = [x+y]_t$
 - $k * [x]_t = [kx]_t$
 - $[x]_t * [y]_t = [x*y]_{2t}$ – needs $2t+1$ shares to reconstruct the secret

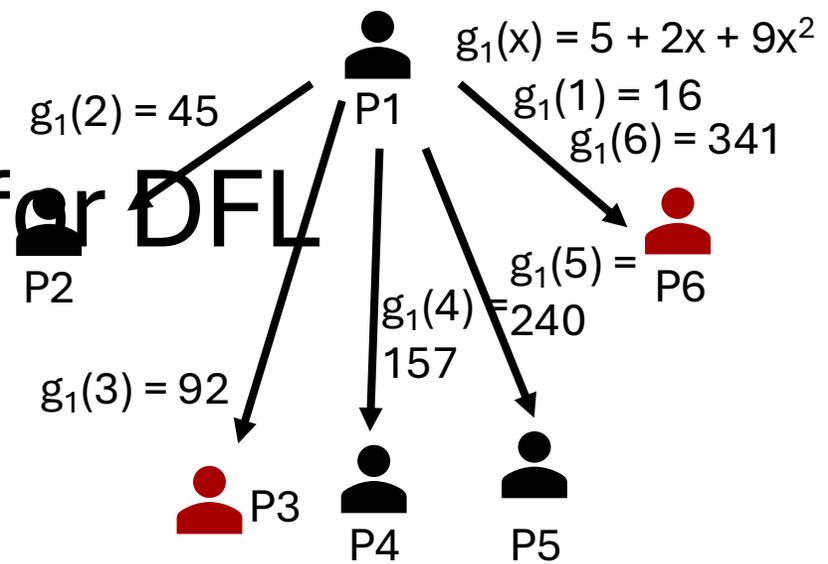
DN Multiplication Protocol

- The parties prepare $([r]_t, [r]_{2t})$, where r is a secret random value.
 - Each party locally computes $[x]_t * [y]_t = [xy]_{2t}$ and sets $[e]_{2t} := [xy]_{2t} + [r]_{2t}$.
 - The parties select a coordinator P_{king} and send $[e]_{2t}$ to him.
 - P_{king} reconstructs e , generates $[e]_t$, and distributes it to the parties.
 - Each party then locally computes $[z]_t = [xy]_t = [e]_t - [r]_t$.
- During this process, any party, including P_{king} , may cheat. This protocol provides only semi-honest security.

Robust and Secure Aggregation for DFL

Secure with Abort

Secure with Abort



The tuple $([x]_t, [y]_t, [r]_t, [r]_{2t}, [e]_{2t}, [e]_t, [z]_t)$ is called the **transcript** of the DN protocol. All parties must verify that every value in the transcript is consistent, i.e., that all corresponding shares are well-formed.

- Recall the example of Shamir secret sharing: the parties must ensure that all (id, share) pairs lie on the same polynomial, e.g., $5 + 2x + 9x^2$.
- However, directly revealing shares would compromise privacy.

Suppose $\{([x^{(i)}]_t, [y^{(i)}]_t, [r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t)\}_{i=1}^m$ are the m transcripts that need to be verified.

1. The parties first run the DN protocol on random inputs $([x^{(0)}]_t, [y^{(0)}]_t)$ to obtain an additional transcript $([x^{(0)}]_t, [y^{(0)}]_t, [r^{(0)}]_t, [r^{(0)}]_{2t}, [e^{(0)}]_{2t}, [e^{(0)}]_t, [z^{(0)}]_t)$
2. Next, the parties generate a public random value λ . Each party locally computes

$$([\alpha]_t, [\beta]_t, [\delta]_t, [\delta]_{2t}, [\eta]_{2t}, [\eta]_t, [\gamma]_t) = \sum_{i=0}^m \lambda^i ([x^{(i)}]_t, [y^{(i)}]_t, [r^{(i)}]_t, [r^{(i)}]_{2t}, [e^{(i)}]_{2t}, [e^{(i)}]_t, [z^{(i)}]_t)$$

3. Each party then reveals its shares of $([\alpha]_t, [\beta]_t, [\delta]_t, [\delta]_{2t}, [\eta]_{2t}, [\eta]_t, [\gamma]_t)$, and the parties check whether all revealed shares lie on the same degree- t or degree- $2t$ polynomial, respectively.

Robust and Secure Aggregation for DFL

Guaranteed Output Delivery

Guaranteed Output Delivery

1. For each share that a sender P_d distributes to a receiver P_i , denote it by $\sigma_i(d)$. All other parties P_v act as **verifiers** for this share.
2. Each verifier P_v prepares an authentication key $\mu_{v \rightarrow i}$, for the receiver P_i . The verifier and the sender P_d then run a protocol to compute an authentication tag $T_{d,i,v} = \mu_{v \rightarrow i} * \sigma_i(d)$, and the result is delivered **only to P_i** (neither P_d nor P_v receives the output. That is, $x_d = \sigma_i(d)$, $x_v = \mu_{v \rightarrow i}$, $y_i = T_{d,i,v}$, and all other inputs and outputs are empty.
3. Whenever $\sigma_i(d)$ is disputed — for example, when all other shares lie on a certain polynomial but this one does not — P_i reveals the corresponding authentication tags to all verifiers. The verifiers then perform a **majority vote** to determine whether the sender P_d or the receiver P_i is cheating.
4. The cheating party is eliminated, and the computation is restarted.

Robust and Secure Aggregation for DFL

Remainder questions and Experimental Results

- During the computation of authentication keys and authentication tags, parties may still behave maliciously.
- When a party is eliminated due to detected misbehavior, the number of remaining parties may fall below $2t+1$, in which case degree- $2t$ secret sharings can no longer be reconstructed.
- In addition to addition and multiplication, protocols are required for other types of operations as well.
- Machine learning computations typically use floating-point numbers, whereas secret sharing schemes operate over fixed-point representations.
- All of the above issues are addressed by specialized secure multiparty computation protocols. This project involves approximately 40 such protocols.

Implementation: Based on the semi-honest DN multiplication protocol in MP-SPDZ, we incorporated additional operation logic and ensured that all operations provide guaranteed output delivery under malicious behavior. The implementation consists of 3,000+ lines of C++ code.



```
(17) Eval
├── (7) Compute
│   ├── (2) Rand (Protocols/ShamirGSZ/get_randoms_dispute)
│   ├── (3) DoubleRand (Protocols/GSZ/get_double_sharing)
│   ├── (6) Mult (Protocols/GSZ/init_mul, prepare_mul, exchange and finalize_mul)
│   └── (4) Refresh (Protocols/GSZ/refresh and refresh_batch)
│       └── (5) PartialMult
├── (8) Check-Refresh (Protocols/GSZ/check_refresh)
│   ├── (2) Rand: to get random [x(0)] and [r~(0)]
│   ├── (4) Refresh: on [x(0)] to get [x~(0)]
│   └── (1) Challenge (Protocols/GSZ/challenge): simplified version: each active party generat
│       Sum transcripts with A, get ([x], [x~], [r~], [e], [o]), then broadcast
│       ├── If [x] is inconsistent: (34) Analyze-Sharing
│       ├── If [r~] in Corr not 0: (9) Check-Rand
│       ├── If active party does not follow, add to Corr
│       └── If Pking disputes with anyone in T, update Dispute
│           └── Otherwise, all parties output ok
├── Verify-PartialMult (Protocols/GSZTransc/verify_pm)
│   ├── (12) De-Linearization (done during transcript collection in Protocols/GSZ)
│   ├── (13) Dimension-Reduction (repeat) (Protocol/GSZTransc/dim_reduction)
│   ├── (10) Extend-PartialMult (Protocol/GSZTransc/ext_pm)
│   ├── (11) Compress (Protocol/GSZTransc/compress)
│   ├── (14) Randomization (Protocol/GSZTransc/randomization)
│   └── (15) Check-SingleMult (Protocol/GSZTransc/check_single_mult)
├── (18) Verify-Sharing (Protocol/GSZTag/verify_sharing): ensure sharings are consistent
│   ├── (2) Rand
│   ├── (1) Challenge
│   └── (34) Analyze-Sharing
├── (30) Tag (Protocol/GSZTag/tag): compute tag for each share, only reconstruct to the share rece
├── (24) Key (Protocol/GSZTag/key): P_u uses twisted sharing to share long-term authentication
├── (20) Key-Distribution (Protocol/GSZTag/key_distribution): dispute-aware twisted sharin
├── (21) Check-Key (Protocol/GSZTag/check_key): check if key is a consistent twisted shari
│   ├── (1) Challenge
│   ├── (22) FL-Key: fault localization if Check-Key fail
│   └── (25) SingleTagComp (Protocol/GSZTag/single_tag_comp): MPC to let P_i reconstruct tag based
├── (27) Check-Tag (Protocol/GSZTag/check_tag): check if tag is a valid sharing
├── (26) P_baseSharing (Protocol/GSZTag/p_base_sharing): generate share masks
├── (25) SingleTagComp: on generated shares masks
│   ├── (1) Challenge
│   ├── (28) FL-Tag: if Check-Tag fails, try to find P_v~P_i dispute
│   └── (29) FL-SingleDealer: if FL-Tag fails, try to P_v~P_d dispute
```

The protocol was successfully executed in a distributed network with nodes located in Utah, Wisconsin, and South Carolina.

Site Site 2 Cluster: Cloudlab Utah

Site Site 3 Cluster: Cloudlab Wisconsin

Site Site 1 Cluster: Cloudlab Clemson

