Secure Multiparty Computation with Lazy Sharing

Shuaishuai Li, **Cong Zhang**, and Dongdai Lin

ACM CCS 2024

(t, n) Secret Sharing

A (t, n) secret sharing scheme:

- Share. For a secret x, output n shares $(x_1, \ldots, x_n)$.
- Recover. Given any $t+1$ shares $\{x_j\}_{j \in J}$ ($|J| = t+1$), output a value x'.

Correctness: For any $(x_1, \ldots, x_n) \leftarrow \mathrm{Share}(x)$ and any $J \subseteq [n]$ with $|J| = t+1$,

$$\mathrm{Recover}(\{x_j\}_{j \in J}) = x.$$

Privacy: For J with $|J| \leq t$, the distribution of $\{x_j\}_{j \in J}$ is independent of x.

Generic MPC from Secret Sharing

A generic n-party MPC against t corrupted parties contains:

- **Input Sharing.** Creat a (t, n)-sharing for each input.
- **Circuit Evaluation.** Evaluate the circuit gate-by-gate. The output of each gate will be a (t, n)-sharing.
- **Output Recovery.** For each output gate, recover the output.

MPC Models

MPC works in two main models:

- *Standard Model.* The parties privide inputs.
- *Client-Server Model.* The parties only have sharings of the inputs.

Main Observation

We observe that in the **standard** model, when sharing an input,

1. *If P_i is the input owner, then creating a (t, n)-sharing is overkill.*
2. *For any t shares containing x_i, they are allowed to contain any information about x.*

Lazy Sharing

Lazy sharing has a relaxed privacy:

$$\textit{For } J \subseteq [n] \backslash \mathcal{L} \textit{ with } |J| \leq t, \textit{ the distribution of } \{x_j\}_{j \in J} \textit{ is independent of } x.$$

Here, $\mathcal{L} \subseteq [n]$ is a parameter. If $\mathcal{L} = \varnothing$, then lazy sharing degenerates into standard sharing.

Application: Using lazy sharing, we can improve efficiency of MPC protocols based on additive or replicated sharing.

Lazy Additive Sharing

When sharing an input x of P_1:

- **Additive Sharing.** P_1 samples $n-1$ random values $x_1, \ldots, x_{n-1}$ and computes $x_n = x - \sum_{j\in[n-1]} x_j$. The sharing is $\langle x \rangle = (x_1, \ldots, x_n)$.
- **Lazy Additive Sharing.** The sharing is simply $\langle x \rangle_{\{1\}} = (x, 0, \ldots, 0)$.

GMW with Additive Sharing

GMW (Goldreich-Micali-Wigderso, STOC 1987) is a foundational generic MPC protocol,

- **Input Sharing.** Each input $x \to$ additive sharing $(x_1, \ldots, x_n)$
- **Circuit Evaluation.** To multiply two sharings $\langle x \rangle = (x_1, \ldots, x_n), \langle y \rangle = (y_1, \ldots, y_n)$,

  For every $(i, j) \in [n]^2$ with $i \neq j$, P_i and P_j additively share $x_i y_j$.

  This requires $n(n-1)$ OLEs.
- **Output Recovery.** To recover $\langle z \rangle$ to P_1, each other party P_j sends z_j to P_1.

LGMW: GMW with Lazy Additive Sharing-Input Sharing

Input Sharing.

Each P_i secret-share its input $x \to \langle x \rangle_{\{i\}}$, where

$$x_j = \begin{cases} x, & \text{if } j = i \\ 0, & \text{if } j \neq i \end{cases}$$

Circuit Evaluation.

To multiply two sharings $\langle x \rangle_{\mathcal{L}_0}, \langle y \rangle_{\mathcal{L}_1} \to \langle xy \rangle_{\mathcal{L}_0 \cup \mathcal{L}_1}$,

For every $(i, j) \in [n]^2 \backslash \mathcal{L}_0 \times \mathcal{L}_1$, $x_i y_j = 0$ *has been an additive sharing* $x_i y_j = 0 + 0$.

The task is that

For every $(i, j) \in \mathcal{L}_0 \times \mathcal{L}_1$ *with* $i \neq j$, P_i *and* P_j *additively share* $x_i y_j$.

This requires only $|\mathcal{L}_0| \cdot |\mathcal{L}_1| - |\mathcal{L}_0 \cap \mathcal{L}_1|$ OLEs.

LGMW: GMW with Lazy Additive Sharing-Output Recovery

Output Recovery.

In GMW: Each output sharing is uniform sharing, meaning that

Any $n-1$ shares are random values. Collecting all the shares are secure.

In LGMW, each output sharing may be not uniform: some share may even be an input!

The parties must invoke a secure sum protocol.

Lazy Replicated Sharing

We focus on $(1, 3)$-replicated sharing. To share an input x of P_1:

- **Replicated Sharing.** P_1 samples three random values x_1, x_2, x_3 subject to $x_1 + x_2 + x_3 = x$. The final sharing is $(x_2, x_3)(x_3, x_1)(x_1, x_2)$.
- **Lazy Replicated Sharing.** P_1 samples two random values x_2, x_3 subject to $x_2 + x_3 = x$. The sharing is $(x_2, x_3)(x_3, 0)(0, x_2)$.

AFLNO with Replicated Sharing

AFLNO (Araki et al., CCS 2016) is fast three-party protocol,

- **Input Sharing.** Each input $x \to$ replicated sharing $(x_2, x_3)(x_3, x_1)(x_1, x_2)$
- **Circuit Evaluation.** To multiply two sharings $(x_2, x_3)(x_3, x_1)(x_1, x_2)$, $(y_2, y_3)(y_3, y_1)(y_1, y_2)$,
  1. Each P_i computes $z_{i-1} = x_{i-1}y_{i-1} + x_{i-1}y_{i+1} + x_{i+1}y_{i-1} + r_i$.
  2. Each P_i sends z_{i-1} to P_{i+1}.
  3. The sharing is $(z_3, z_2)(z_1, z_3)(z_2, z_1)$.
- **Output Recovery.** To recover $(z_3, z_2)(z_1, z_3)(z_2, z_1)$ to P_1, P_2 or P_3 sends z_1 to P_1.

AFLNO with Improved Input Sharing

In AFLNO, if P_1 wants to secret-share its input x, it must send 4 elements: (x_3, x_1) to P_2 and (x_1, x_2) to P_3.

Using lazy replicated sharing, the communication is 2 elements:

P_1 samples x_2, x_3 s.t. $x_2 + x_3 = x$: send x_3 to P_2 and x_2 to P_3.

The sharing is $(x_2, x_3)(x_3, 0)(0, x_2)$.

Performance: GMW and LGMW

Circuit	n	Computation (s)			Runtime (ms)			Communication (MB)			Throughput (gates/s)		
		GMW	LGMW	Imp	GMW	LGMW	Imp	GMW	LGMW	Imp	GMW	LGMW	Imp
Product	6	3.805	0.376	10.12×	248.9	68.7	3.62×	1.465	0.147	9.97×	2.01	7.28	3.62×
	8	9.940	0.683	14.55×	487.6	71.2	6.85×	3.829	0.274	13.97×	1.44	9.83	6.83×
	10	20.729	1.110	18.67×	813.5	125.8	6.47×	7.911	0.440	17.98×	1.11	7.15	6.44×
Inner Product	6	2.318	0.073	31.75×	151.6	11.6	13.07×	0.879	0.029	30.31×	3.30	43.10	13.06×
	8	5.695	0.096	59.32×	279.4	12.1	23.09×	2.188	0.039	56.10×	2.51	57.85	23.05×
	10	11.407	0.125	91.26×	456.8	12.7	35.97×	4.395	0.050	87.90×	1.97	70.87	35.97×
Chain	6	4.425	0.490	9.03×	276.3	96.7	2.86×	1.465	0.147	9.97×	3.62	10.34	2.86×
	8	11.614	0.846	13.73×	556.8	165.9	3.36×	3.829	0.274	13.97×	2.51	8.44	3.36×
	10	24.315	1.357	17.92×	953.6	245.6	3.88×	7.912	0.440	17.98×	1.89	7.33	3.88×

Table 6: Comparision of GMW and LGMW for computing the product, inner product, and chain circuits. "Imp" is an abbreviation for "improvement". The throughput is computed as the number of gates processed by the protocol per second.

Figure: Performance of GMW and LGMW.

Performance: AFLNO and LAFLNO

Circuit	Computation (ms)			Runtime (ms)			Communication (byte)			Throughput (gates/ms)		
	AFLNO	LAFLNO	Imp	AFLNO	LAFLNO	Imp	AFLNO	LAFLNO	Imp	AFLNO	LAFLNO	Imp
Sum	0.569	0.434	1.31×	0.174	0.156	1.12×	104	56	1.86×	11.49	12.82	1.12×
Product	0.752	0.633	1.19×	0.276	0.253	1.09×	152	104	1.46×	7.25	7.91	1.09×
Inner Product	1.567	1.256	1.25×	0.345	0.313	1.10×	272	176	1.55×	14.49	15.97	1.10×
Chain	1.418	1.104	1.28×	0.385	0.344	1.12×	216	136	1.59×	10.39	11.63	1.12×

Table 7: Comparision of AFLNO and LAFLNO for computing the sum, product, inner product, and chain circuits. "Imp" is an abbreviation for "improvement". The throughput is computed as the number of gates processed by the protocol per second.

Figure: Performance of AFLNO and LAFLNO.

Q&A

Any Question?

ePrint: https://eprint.iacr.org/2024/1347