New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning[a]

Presented at S&P 2019

Ivan Damgård[1] Daniel Escudero[1] Tore Frederiksen[2] Marcel Keller[3] Peter Scholl[1] Nikolaj Volgushev[2]

[1]Aarhus University, Denmark

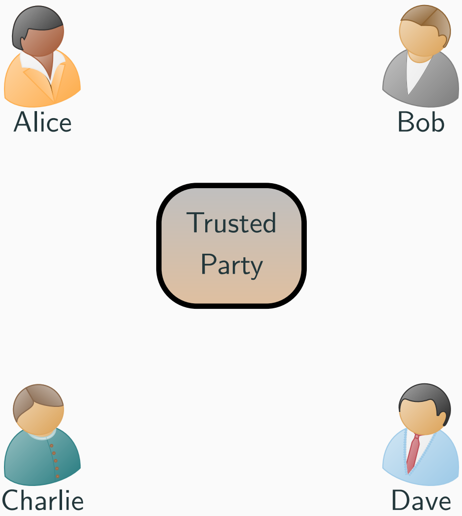
[2]Alexandra Institute, Denmark

[3]Data61, CSIRO, Australia

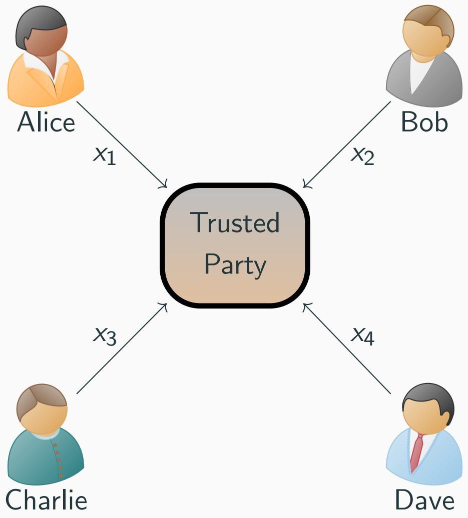
[a]Supported by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme under grant agreements No 669255 (MPCPRO), No 731583 (SODA) and the Danish Independent Research Council under Grant-ID DFF6108-00169 (FoCC).

Introduction

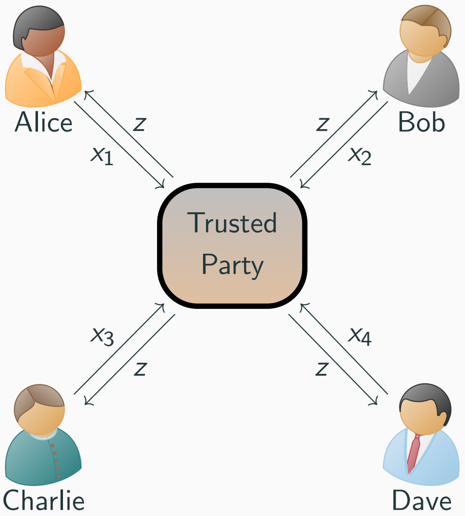
MPC



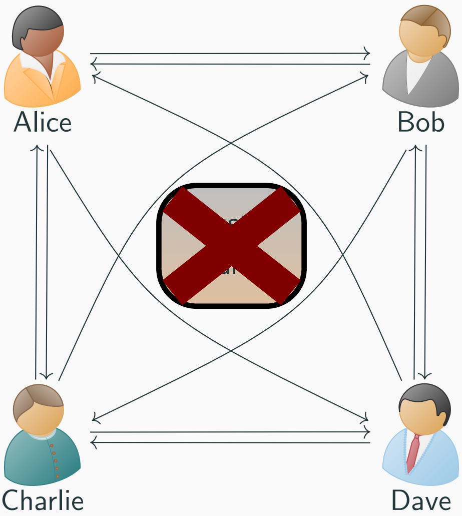
MPC



MPC



MPC



Many different approaches to MPC

Circuits over $\mathbb{F}_2$

- Garbled Circuits
- BMR
- GMW
- $\cdots$

Circuits over $\mathbb{F}_p$

- BGW
- BeDOZa
- SPDZ
- MASCOT
- $\cdots$

Many different approaches to MPC

Circuits over $\mathbb{Z}_{2^k}$

- Sharemind, *Bogdanov et al. ESORICS 2008*
- Replicated SS with Active Security, *Furukawa et al. EUROCRYPT 2017*
- Passive-To-Active Compiler, *Damgård et al. CRYPTO 2018*
- SPD$\mathbb{Z}_{2^k}$, *Cramer et al. CRYPTO 2018.*

Our Contribution

New sub-protocols for SPD$\mathbb{Z}_{2^k}$

We expand SPD$\mathbb{Z}_{2^k}$ with a series of sub-protocols to enhance the potential range of applications.

- Arithmetic-Binary share conversions
- Random-bit generation
- Bit-decomposition
- Secure truncation, comparison and equality check.

SPD$\mathbb{Z}_{2^k}$ implementation

We implement the SPD$\mathbb{Z}_{2^k}$ protocol in Java, as part of the *FRamework for Efficient Secure COmputation* (FRESCO).

- Our implementation contains several optimizations that can be of independent interest.
- In the microbenchmarks we observe several improvements with respect to other protocols over fields.

Applications to Secure Machine Learning

We illustrate the benefits of our techniques by performing certain ML tasks in SPD$\mathbb{Z}_{2^k}$ and observe several improvements with respect to other protocols over fields. We consider:

- Secure evaluation of Decision Trees
- Secure evaluation of Support Vector Machines

SPD$\mathbb{Z}_{2^k}$

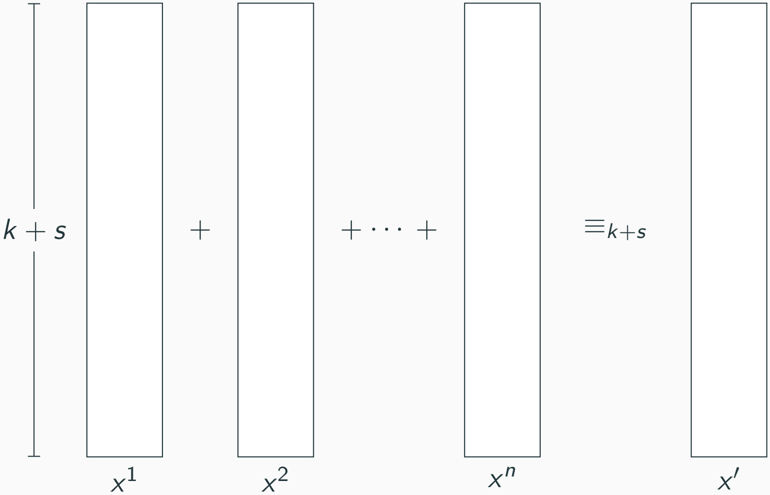
SPD$\mathbb{Z}_{2^k}$ in a nutshell

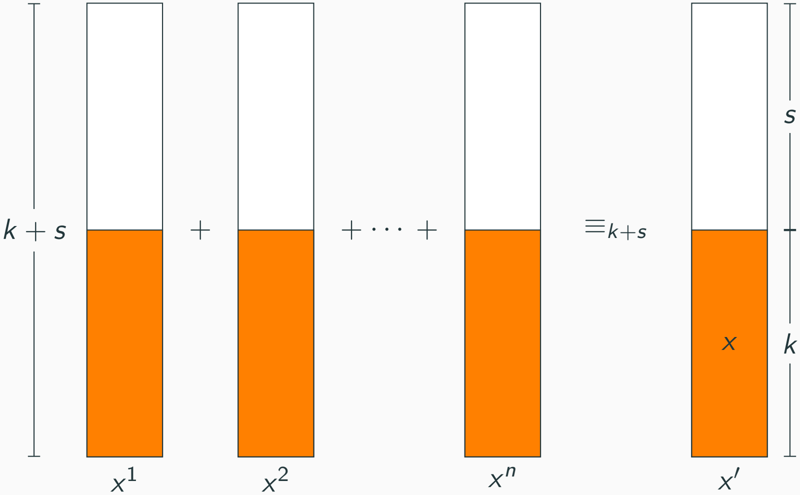
Additive Authenticated Secret-Sharing over $\mathbb{Z}_{2^k}$

$x \in \mathbb{Z}_{2^k}$ is shared, denoted by $[x]_{2^k}$, if

- Each P_i has $x^i, \alpha^i, m^i \in \mathbb{Z}_{2^{k+s}}$
- $\sum x^i \equiv_{k+s} x'$ with $x' \equiv_k x$
- $\sum \alpha^i \equiv_{k+s} \alpha$, where $\alpha \in \mathbb{Z}_{2^s}$ is a random global key
- $\sum m^i \equiv_{k+s} \alpha \cdot x'$.

$x \equiv y \bmod 2^\ell$ is abbreviated by $x \equiv_\ell y$


$k + s$
$+$
$+ \cdots +$
$\equiv_{k+s}$
x^1
x^2
x^n
x'


$k+s$
$+$
$+\cdots+$
$\equiv_{k+s}$
s
x
k
x^1
x^2
x^n
x'

Secure computation with preprocessing

Input phase

$$[x_i]_{2^k} = \underbrace{(x_i - r_i)}_{\text{broadcast}} + [r_i]_{2^k}$$

where x_i are the inputs and $(r_i, [r_i]_{2^k})$ is preprocessed.

Addition gates

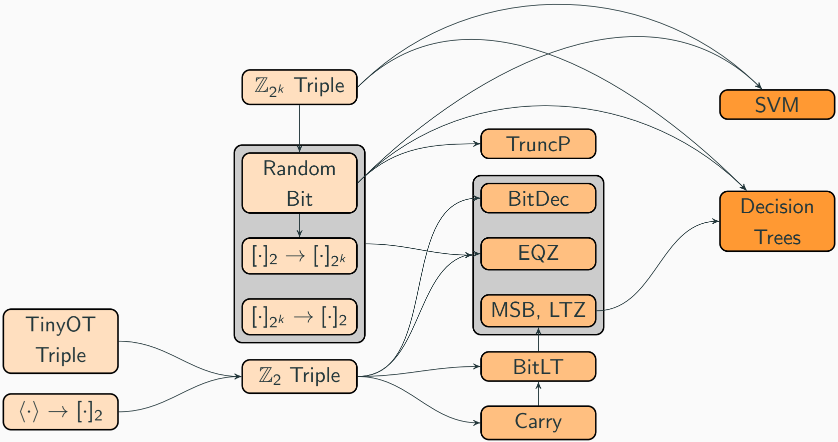
$$[x + y]_{2^k} = [x]_{2^k} + [y]_{2^k}$$

Multiplication gates

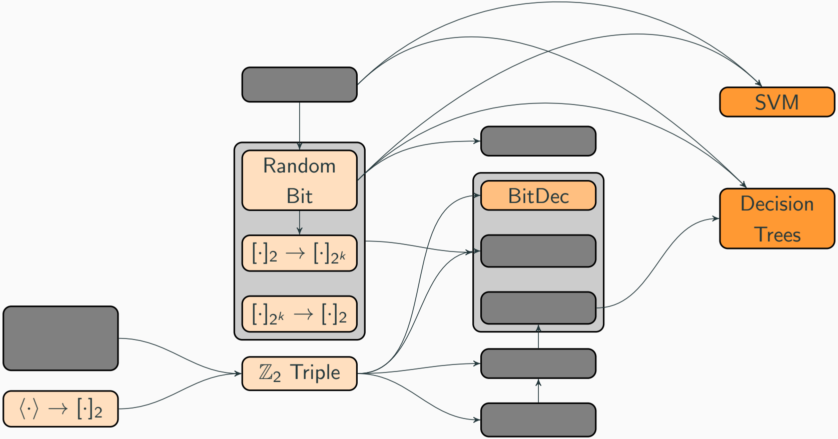
$$[x \cdot y]_{2^k} = [c]_{2^k} + \underbrace{(x - a)}_{\text{open}} \cdot [b]_{2^k} + \underbrace{(y - b)}_{\text{open}} \cdot [a]_{2^k} + \underbrace{(x - a)}_{\text{open}} \underbrace{(y - b)}_{\text{open}}$$

where $([a]_{2^k}, [b]_{2^k}, [c]_{2^k})$ is preprocessed with $c = a \cdot b$.

Primitives for MPC Modulo 2^k



$\mathbb{Z}_{2^k}$ Triple
SVM
TruncP
Random Bit
BitDec
Decision Trees
$[\cdot]_2 \to [\cdot]_{2^k}$
EQZ
$[\cdot]_{2^k} \to [\cdot]_2$
MSB, LTZ
TinyOT Triple
$\mathbb{Z}_2$ Triple
BitLT
$\langle\cdot\rangle \to [\cdot]_2$
Carry


SVM
Random Bit
Decision Trees
BitDec
$[\cdot]_2 \to [\cdot]_{2^k}$
$[\cdot]_{2^k} \to [\cdot]_2$
$\mathbb{Z}_2$ Triple
$\langle\cdot\rangle \to [\cdot]_2$

Generating Random Bits $[b]_{2^k}$ (Intuition)

Ideal Protocol

1. Sample $[r]_{2^k}$ at random and let $[a]_{2^k} = [r^2]_{2^k}$.
2. Open a. Let c be *some* square root of a.
3. Compute $[d]_{2^k} = c^{-1}[r]_{2^k}$.
   - Now d is a random square root of 1, so $d \in_R \{-1, +1\}$.
4. Output $[b]_{2^k}$, where $b = (d + 1)/2$.

Generating Random Bits $[b]_{2^k}$ (Intuition)

Ideal Protocol

1. Sample $[r]_{2^k}$ at random and let $[a]_{2^k} = [r^2]_{2^k}$.
2. Open a. Let c be *some* square root of a.
3. Compute $\cancel{[d]_{2^k} = c^{-1}[r]_{2^k}.}$
   - Now d is a random square root of 1, so $\cancel{d \in_R \{-1, +1\}.}$
4. Output $[b]_{2^k}$, where $\cancel{b = (d+1)/2.}$

Generating Random Bits $[b]_{2^k}$ (Intuition)

Actual Protocol

1. Sample $[r]_{2^{k+2}}$ at random, where r is odd, and let $[a]_{2^{k+2}} = [r^2]_{2^{k+2}}$.
2. Open a. Let c be *some* square root of a.
3. Compute $[d]_{2^{k+2}} = c^{-1}[r]_{2^{k+2}}$
   - Now d is a random square root of 1 mod 2^{k+2}, so $d \in_R \{-1, +1, -1 + 2^{k+1}, +1 + 2^{k+1}\}$.
4. Output $[b]_{2^k}$, where $b \equiv_k (d + 1)/2$.

Share Conversions

$[b]_{2^k} \to [b]_2$

Local reduction modulo 2.[a]

[a] In fact, it is reduction modulo 2^{s+1} for the extra s "MAC" bits.

$[b]_2 \to [b]_{2^k}$

1. Sample a random bit $[r]_{2^k}$ $(r \in \mathbb{Z}_2)$
2. Convert $[r]_{2^k}$ to $[r]_2$.
3. Open $[c] = [b]_2 \oplus [r]_2$
4. Output $[b]_{2^k} = [r]_{2^k} + [c]_{2^k} - 2[r]_{2^k}[c]_{2^k}$

Bit Decomposition: $[x]_{2^k} \to ([x_0]_{2^k}, \ldots, [x_{k-1}]_{2^k})$

1. Sample random bits $[r_0]_{2^k}, \ldots, [r_{k-1}]_{2^k}$ and let $[r]_{2^k} = \sum_{i=0}^{k-1} 2^i [r_i]_{2^k}$.
2. Compute $[a]_{2^k} = [x]_{2^k} - [r]_{2^k}$ and open a.

4. Compute the binary circuit

$$([x_0]_{2^k}, \ldots, [x_{k-1}]_{2^k}) = \mathrm{ADD}\left((a_0, \ldots, a_{k-1}), ([r_0]_{2^k}, \ldots, [r_{k-1}]_{2^k})\right).$$

Bit Decomposition: $[x]_{2^k} \to ([x_0]_{2^k}, \ldots, [x_{k-1}]_{2^k})$

1. Sample random bits $[r_0]_{2^k}, \ldots, [r_{k-1}]_{2^k}$ and let $[r]_{2^k} = \sum_{i=0}^{k-1} 2^i [r_i]_{2^k}$.
2. Compute $[a]_{2^k} = [x]_{2^k} - [r]_{2^k}$ and open a.
3. Convert $([r_0]_{2^k}, \ldots, [r_{k-1}]_{2^k})$ to $([r_0]_2, \ldots, [r_{k-1}]_2)$.
4. Compute the binary circuit

$$([x_0]_2, \ldots, [x_{k-1}]_2) = \mathrm{ADD}\left((a_0, \ldots, a_{k-1}), ([r_0]_2, \ldots, [r_{k-1}]_2)\right).$$

5. Convert the result $([x_0]_2, \ldots, [x_{k-1}]_2)$ to $([x_0]_{2^k}, \ldots, [x_{k-1}]_{2^k})$.

Implementation and Benchmarks

Online Phase - Micro Operations

Throughput in elements per second for the online phase of micro operations over 1 Gbps network. The factor columns express the runtime improvement factor of SPD$\mathbb{Z}_{2^k}$ over SPDZ in FRESCO.

	$k = 32$			$k = 64$		
	SPD$\mathbb{Z}_{2^k}$ ($\sigma = 26$)	SPDZ ($\sigma = 26$)	**Factor**	SPD$\mathbb{Z}_{2^k}$ ($\sigma = 57$)	SPDZ ($\sigma = 57$)	**Factor**
Multiplication	687041	141346	**4.9x**	522258	114071	**4.6x**
Equality	15334	3213	**4.8x**	6902	1282	**5.4x**
Comparison	9153	1769	**5.2x**	4514	756	**6.0x**

Online Phase for SVMs Evaluation

Online phase benchmarking of SVM evaluation over 1 Gbps network. The factor columns express the runtime improvement factor of SPD$\mathbb{Z}_{2^k}$ over SPDZ in FRESCO. Times are in milliseconds per sample.

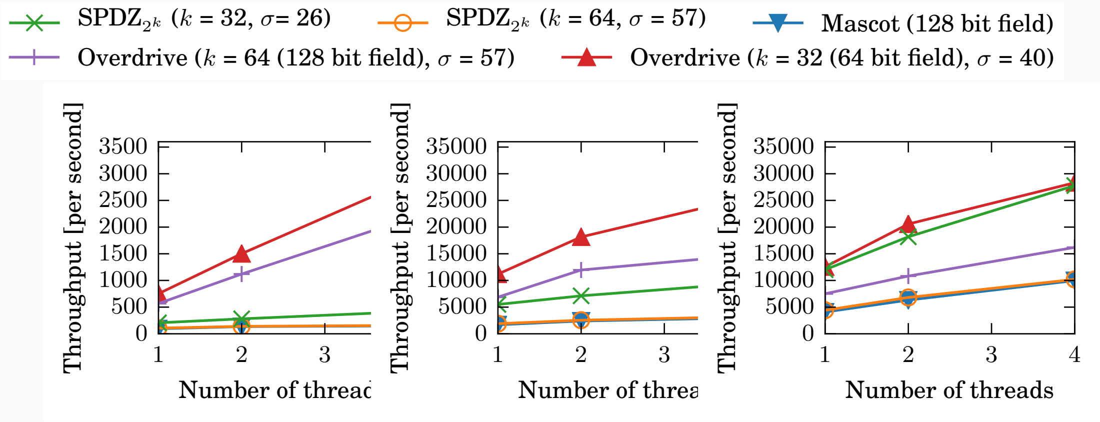
			$k = 32,\ \sigma = 26$			$k = 64,\ \sigma = 57$		
Dataset	Num. Classes, Features	Batch Size	SPD$\mathbb{Z}_{2^k}$	SPDZ	**Factor**	SPD$\mathbb{Z}_{2^k}$	SPDZ	**Factor**
CIFAR	10, 2048	1	82 ms	214 ms	**2.6x**	99 ms	255 ms	**2.6x**
MIT	67, 2048	1	379 ms	1318 ms	**3.5x**	499 ms	1582 ms	**3.2x**
ALOI	463, 128	1	242 ms	857 ms	**3.5x**	362 ms	1312 ms	**3.6x**
CIFAR	10, 2048	5	39 ms	168 ms	**4.3x**	57 ms	209 ms	**3.7x**
MIT	67, 2048	5	225 ms	1101 ms	**4.9x**	294 ms	1428 ms	**4.9x**
ALOI	463, 128	5	162 ms	741 ms	**4.6x**	244 ms	1220 ms	**5.0x**

Online Phase for Decision Trees Evaluation

Online phase benchmarking of evaluation of decision trees over 1 Gbps network. The factor columns express the runtime improvement factor of SPD$\mathbb{Z}_{2^k}$ over SPDZ in FRESCO. Times are in milliseconds per sample.

			$k = 32,\ \sigma = 26$			$k = 64,\ \sigma = 57$		
Dataset	Depth, Num. Features	Batch Size	SPD$\mathbb{Z}_{2^k}$	SPDZ	**Factor**	SPD$\mathbb{Z}_{2^k}$	SPDZ	**Factor**
Hill Valley	3, 100	1	21 ms	24 ms	**1.2x**	26 ms	34 ms	**1.3x**
Spambase	6, 57	1	48 ms	104 ms	**2.2x**	56 ms	128 ms	**2.3x**
Diabetes	9, 8	1	80 ms	215 ms	**2.7x**	122 ms	443 ms	**3.6x**
Hill Valley	3, 100	5	6 ms	10 ms	**1.7x**	7 ms	15 ms	**2.1x**
Spambase	6, 57	5	14 ms	40 ms	**2.9x**	17 ms	68 ms	**4.0x**
Diabetes	9, 8	5	41 ms	185 ms	**4.5x**	78 ms	376 ms	**4.8x**

Triple Generation Throughput



(a) WAN (50 Mbps, 100 ms latency) **(b)** LAN (1 Gbps, 0.1 ms latency) **(c)** LAN (10 Gbps, 0.1 ms latency)

AARHUS UNIVERSITY

Communication Complexity

Total theoretical communication complexity counted in (kilo-, mega-, giga-) bytes for the two-party case. Values for SPDZ are based on Overdrive in Low Gear. For bit triples we use the optimized TinyOT WRK protocol.

	$k = 32,\ \sigma = 26$				$k = 64,\ \sigma = 57$			
	SPD$\mathbb{Z}_{2^k}$		SPDZ (64 bit field)		SPD$\mathbb{Z}_{2^k}$		SPDZ (128 bit field)	
	Preprocessing	**Online**	Preprocessing	**Online**	Preprocessing	**Online**	Preprocessing	**Online**
Comparison	627 KB	46 B	148 KB	1.89 KB	3.58 MB	94 B	508 KB	7.78 KB
Equality	486 KB	24 B	107 KB	1.01 KB	3.08 MB	48 B	366 KB	3.97 KB
DTree (Diabetes)	209 MB	131 KB	40.8 MB	705 KB	1.10 GB	262 KB	110 MB	2.37 MB
SVM (ALOI)	908 MB	1.44 KB	139 MB	3.24 MB	4.06 GB	2.88 MB	341 MB	8.29 MB

Conclusions

- We implemented the SPD$\mathbb{Z}_{2^k}$ protocol along with practical primitives for MPC mod 2^k.
- We saw up to a 5-fold improvement in computation for various tasks, and up to a 85-fold reduction in online communication costs for secure comparison, as compared to the field setting.

Future Work

- Close the gap for the preprocessing.
- Expand the range of applications for computation modulo 2^k.

Thank you!

Appendix

How to Compute the Binary Circuit?

SPD$\mathbb{Z}_{2^k}$ does not support triple generation for $k = 1$.

Share Conversion $\langle x \rangle \to [x]_2$

Let $\langle \cdot \rangle$ denote any authenticated sharing mechanism over $\mathbb{Z}_2$ (e.g. TinyOT shares).

- Input: Sharings $\langle x_1 \rangle, \ldots, \langle x_m \rangle$.
- Output: Binary sharings $[x_1]_2, \ldots, [x_m]_2$.

1. Sample s additional random shared bits $\langle r_1 \rangle, \ldots, \langle r_s \rangle$.

1. Sample s additional random shared bits $\langle r_1 \rangle, \ldots, \langle r_s \rangle$.
2. Each party P_i inputs the shares $x_1^i, \ldots, x_m^i, r_1^i, \ldots, r_s^i$ to obtain $[x_1^i], \ldots, [x_m^i], [r_1^i], \ldots, [r_s^i]$

1. Sample s additional random shared bits $\langle r_1 \rangle, \ldots, \langle r_s \rangle$.
2. Each party P_i inputs the shares $x_1^i, \ldots, x_m^i, r_1^i, \ldots, r_s^i$ to obtain $[x_1^i], \ldots, [x_m^i], [r_1^i], \ldots, [r_s^i]$
3. Parties sum these shares to obtain (possibly incorrect) sharings $[x_1]_2, \ldots, [x_m]_2$ and $[r_1]_2, \ldots, [r_s]_2$.

1. Sample s additional random shared bits $\langle r_1 \rangle, \ldots, \langle r_s \rangle$.
2. Each party P_i inputs the shares $x_1^i, \ldots, x_m^i, r_1^i, \ldots, r_s^i$ to obtain $[x_1^i], \ldots, [x_m^i], [r_1^i], \ldots, [r_s^i]$
3. Parties sum these shares to obtain (possibly incorrect) sharings $[x_1]_2, \ldots, [x_m]_2$ and $[r_1]_2, \ldots, [r_s]_2$.
4. Sample $m \cdot s$ random bits $\chi_{i,j} \leftarrow_R \{0, 1\}$, for $i = 1, \ldots, m$ and $j = 1, \ldots, s$, using a coin-tossing protocol.

1. Sample s additional random shared bits $\langle r_1 \rangle, \ldots, \langle r_s \rangle$.
2. Each party P_i inputs the shares $x_1^i, \ldots, x_m^i, r_1^i, \ldots, r_s^i$ to obtain $[x_1^i], \ldots, [x_m^i], [r_1^i], \ldots, [r_s^i]$
3. Parties sum these shares to obtain (possibly incorrect) sharings $[x_1]_2, \ldots, [x_m]_2$ and $[r_1]_2, \ldots, [r_s]_2$.
4. Sample $m \cdot s$ random bits $\chi_{i,j} \leftarrow_R \{0,1\}$, for $i = 1, \ldots, m$ and $j = 1, \ldots, s$, using a coin-tossing protocol.
5. For each j, open $[r_j]_2 \oplus \sum_{i=1}^m \chi_{i,j} \cdot [x_i]_2$ as y_j and $\langle r_j \rangle_2 \oplus \sum_{i=1}^m \chi_{i,j} \cdot \langle x_i \rangle_2$ as y_j'.

1. Sample s additional random shared bits $\langle r_1 \rangle, \ldots, \langle r_s \rangle$.
2. Each party P_i inputs the shares $x_1^i, \ldots, x_m^i, r_1^i, \ldots, r_s^i$ to obtain $[x_1^i], \ldots, [x_m^i], [r_1^i], \ldots, [r_s^i]$
3. Parties sum these shares to obtain (possibly incorrect) sharings $[x_1]_2, \ldots, [x_m]_2$ and $[r_1]_2, \ldots, [r_s]_2$.
4. Sample $m \cdot s$ random bits $\chi_{i,j} \leftarrow_R \{0, 1\}$, for $i = 1, \ldots, m$ and $j = 1, \ldots, s$, using a coin-tossing protocol.
5. For each j, open $[r_j]_2 \oplus \sum_{i=1}^m \chi_{i,j} \cdot [x_i]_2$ as y_j and $\langle r_j \rangle_2 \oplus \sum_{i=1}^m \chi_{i,j} \cdot \langle x_i \rangle_2$ as y_j'.
6. Check that $y_j = y_j'$ for all j. If not, abort.

1. Sample s additional random shared bits $\langle r_1 \rangle, \ldots, \langle r_s \rangle$.
2. Each party P_i inputs the shares $x_1^i, \ldots, x_m^i, r_1^i, \ldots, r_s^i$ to obtain $[x_1^i], \ldots, [x_m^i], [r_1^i], \ldots, [r_s^i]$
3. Parties sum these shares to obtain (possibly incorrect) sharings $[x_1]_2, \ldots, [x_m]_2$ and $[r_1]_2, \ldots, [r_s]_2$.
4. Sample $m \cdot s$ random bits $\chi_{i,j} \leftarrow_R \{0, 1\}$, for $i = 1, \ldots, m$ and $j = 1, \ldots, s$, using a coin-tossing protocol.
5. For each j, open $[r_j]_2 \oplus \sum_{i=1}^{m} \chi_{i,j} \cdot [x_i]_2$ as y_j and $\langle r_j \rangle_2 \oplus \sum_{i=1}^{m} \chi_{i,j} \cdot \langle x_i \rangle_2$ as y'_j.
6. Check that $y_j = y'_j$ for all j. If not, abort.
7. Output the sharings $[x_1]_2, \ldots, [x_m]_2$.