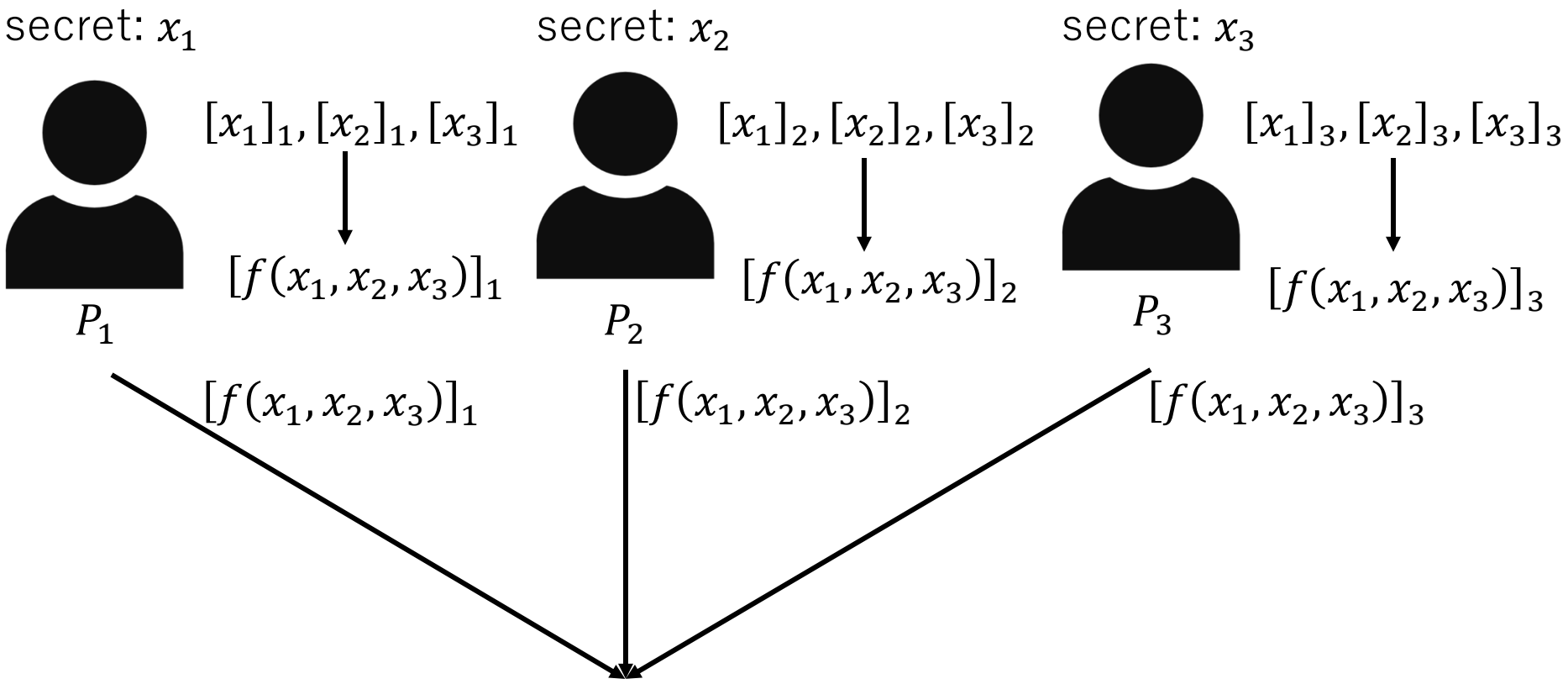
ProVerifによる線形秘密分散の形式化とSPDZの検証への応用

JSIAM年会 FAIS 3C-4-3 @東京理科大学

○**渡部 翔**，三田 翔平，米山 一樹

茨城大学

秘密分散法に基づく秘密計算



$f(x_1, x_2, x_3)$ （x_1, x_2, x_3を明かさずに出力する）

線形秘密分散

- $[x]_i + \varepsilon = [x + \varepsilon]_i$（シェアと定数の加算）
- $[x]_i + [y]_i = [x + y]_i$（シェアとシェアの加算）
- $[x]_i * \varepsilon = [x * \varepsilon]_i$（シェアと定数の乗算）

ローカル計算可能

- $[x]_i * [y]_i = [x * y]_i$（シェアとシェアの乗算）

パーティ間通信が必要

↓

加算と乗算の組み合わせから任意の関数の計算が可能

我々のモチベーション

- 線形秘密分散ベースの秘密計算の実用化が進む
  - ISO/IEC 4922-2*による国際標準化 (2024)
  - 今後、様々な応用プロトコルの社会実装が期待される
- 基盤技術である線形秘密分散や
  SPDZ[DPSZ12]に代表される基盤プロトコルの
  形式化と検証手法を提案
  →将来的な応用プロトコルの検証に役立てる

* https://www.iso.org/standard/80514.html

[DPSZ12] I. Damgård, V. Pastro, N. Smart, S. Zakarias, “Multiparty Computation from Somewhat Homomorphic Encryption”, CRYPTO2012.

本研究の貢献

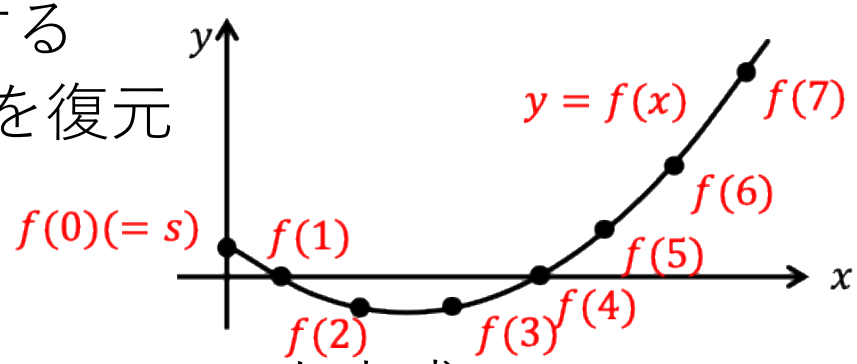
- ProVerifによる線形秘密分散の初めての形式化
  - $n = 3$に固定して形式化（ProVerifでは任意のパーティ数を扱えない）
  - Shamir(n-out-of-n)秘密分散法 / 加法型秘密分散法
- 応用：SPDZの出力フェーズの検証
  - Authenticated Shareを用いる
  - 演算結果出力フェーズにおける
    コラプトパーティ検知の安全性・正当性の検証
- ProVerifで線形性を扱うにあたり
  表現できること・できないことの検討と整理

アウトライン

- 秘密分散法
- シェアおよび定数による演算の線形性の形式化
- 秘密分散・加算・乗算の安全性／正当性の検証
- 応用：SPDZの出力フェーズの検証

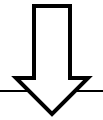
秘密分散法について

- Shamirの秘密分散法（(k,n)しきい値型秘密分散法）
  - 定数項がs(秘密情報)である$k-1$次多項式$f(x)$を適当に生成
    →$f(x)$上の任意のn個の点をシェアとする
  - シェアをk個集めて多項式補間によりsを復元
- 加法型秘密分散法
  - メッセージ空間$\mathcal{M}$ =シェア空間$\mathcal{S} = \mathbb{Z}_q$
  - $s = s_1 + s_2 + \cdots + s_n \bmod q$となるようにシェア$s_i$を生成
- 演算における相違点：
  シェアと定数の加(減)算(Shamir法は全パーティ、加法は1パーティ)



秘密分散のProVerifによる形式化

- 秘密情報：x
- 秘密分散：$x \rightarrow [x]_1, [x]_2, [x]_3$
- 秘密情報の復元：$x' \leftarrow \mathrm{reconst}([x]_1, [x]_2, [x]_3)$
  (3つのシェアが集まらなければ復元はできない(しきい値：3))



```
type unshare. (* 定数 *)
type share. (* シェア *)
fun share1(unshare): share.
fun share2(unshare): share.
fun share3(unshare): share.
letfun secretsharing(s:unshare) = (share1(s),share2(s),share3(s)).
reduc forall x:unshare; recon(share1(x),share2(x),share3(x))=x.
```

演算の線形性の形式化

- シェアと定数の線形性
  - $[x]_i + \varepsilon = [x + \varepsilon]_i$ (シェアと定数の加(減)算)
  - $[x]_i + [y]_i = [x + y]_i$ (シェアとシェアの加(減)算)
  - $[x]_i * \varepsilon = [x * \varepsilon]_i$ (シェアと定数の乗(除)算)

この一般的な変換は
Shamir法のみ

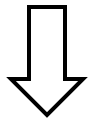
可換性を扱えないため
順序付きの前提

⇩

```
fun addition_share_unshare(share,unshare): share. (* 定数+シェア→シェア *)
fun addition_share(share,share): share. (* シェア+シェア→シェア *)
fun multiplication_share_unshare(share,unshare): share. (* 定数×シェア→シェア *)
```

演算の線形性の形式化

```
fun addition_share_unshare(share,unshare): share. (* 定数+シェア→シェア *)
fun addition_share(share,share): share. (* シェア+シェア→シェア *)
fun multiplication_share_unshare(share,unshare): share. (* 定数×シェア→シェア *)
```



```
equation
forall a,b:unshare; addition_share_unshare(share1(a),b) = share1(addition(a,b));
forall c,d:unshare; addition_share(share1(c),share1(d)) = share1(addition(c,d));
forall e,f:unshare; multiplication_share_unshare(share1(e),f) =          share1(multiplication(e,f)).
```

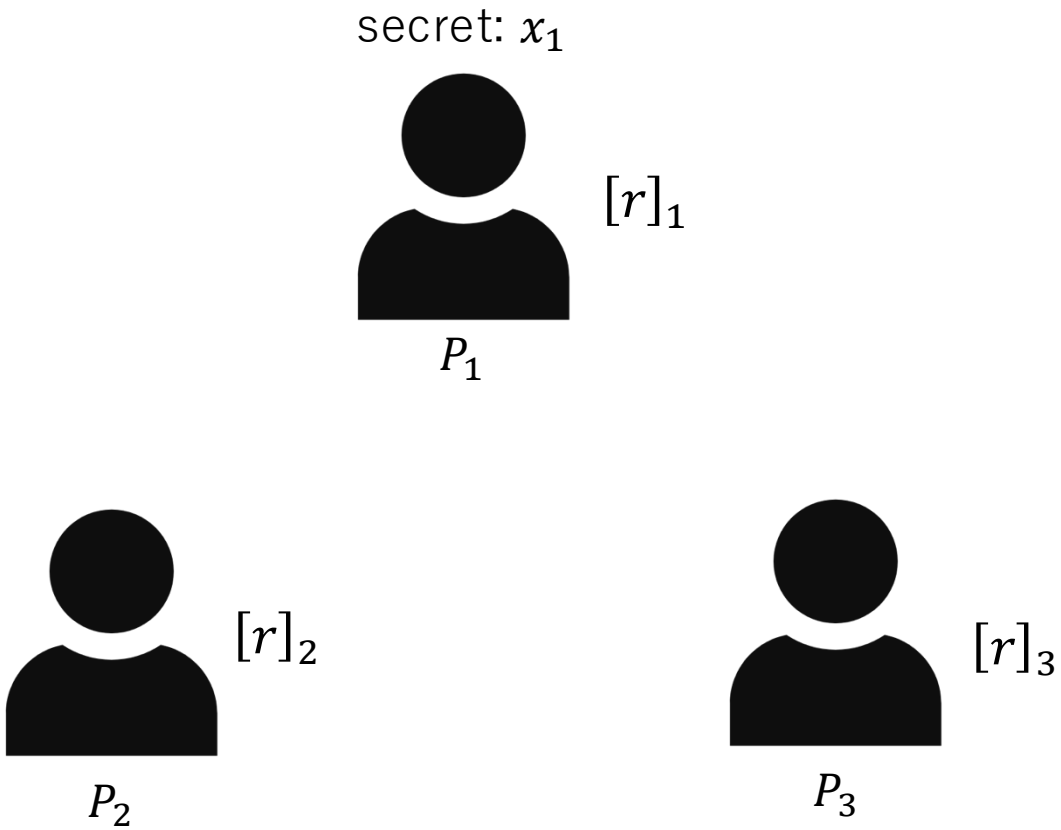
※他のパーティ(P_2, P_3)用にも同様に等式を宣言

加法型秘密分散法の場合

- シェアと定数の加(減)算は特定の1パーティのみ行う
  $([x]_i + \varepsilon = [x + \varepsilon]_i)$
- 復元アルゴリズムにルール付けを行い形式化

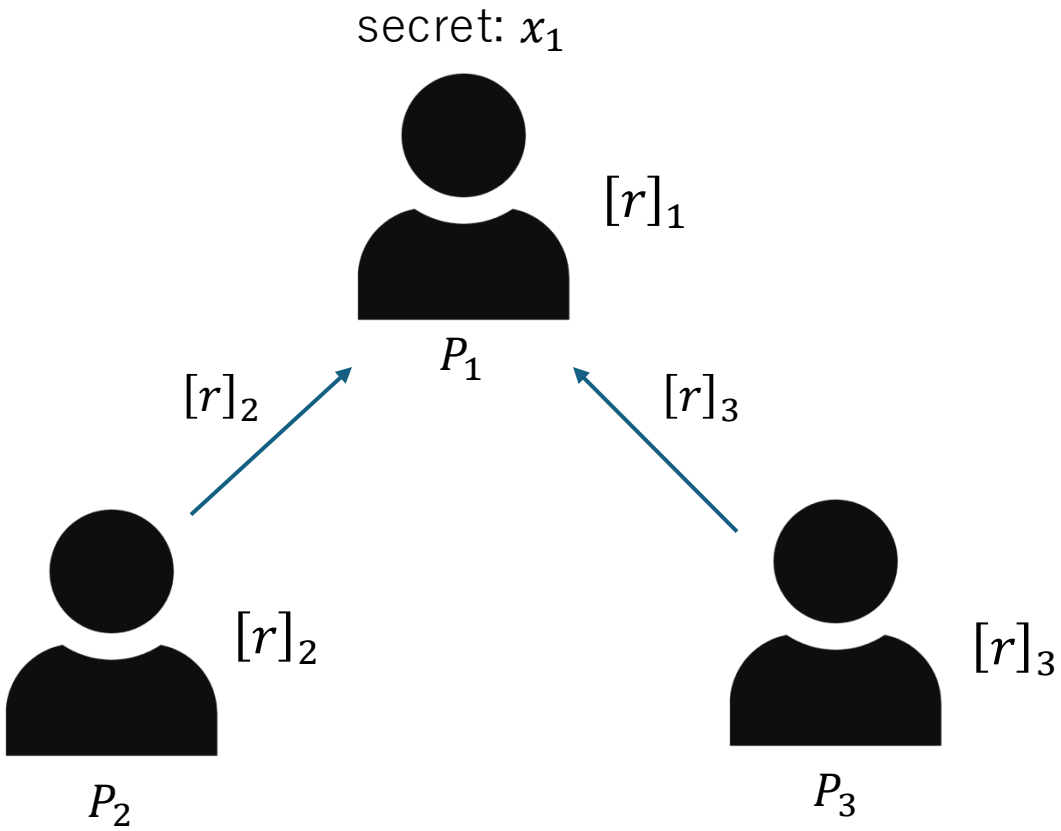
```
fun recon(share,share,share): unshare
reduc forall x:unshare; recon(share1(x),share2(x),share3(x))=x
otherwise forall x,y:unshare;  recon(share1(addition(x,y)),share2(x),share3(x))=addition(x,y).
```

秘密分散の方法

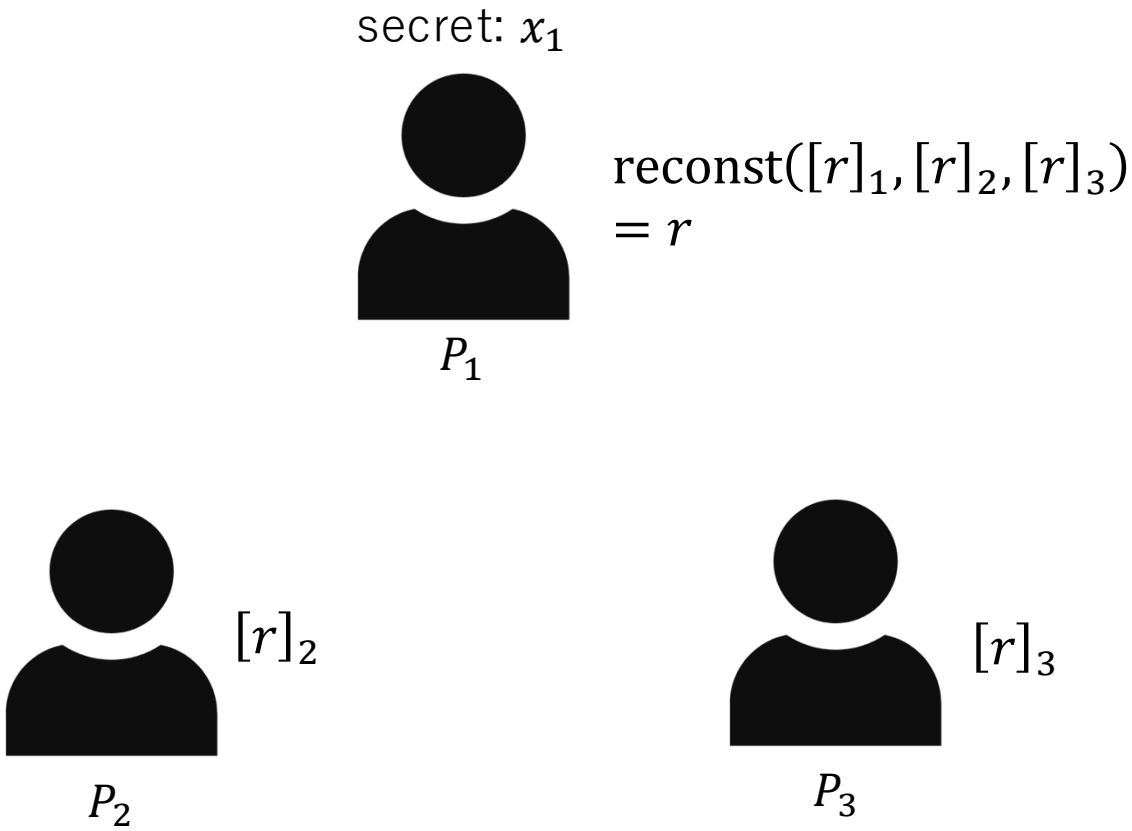


P_1の秘密情報x_1をP_2, P_3にシェアとして分配する
$[r]_i$：事前に各パーティに共有されている乱数のシェア

秘密分散の方法

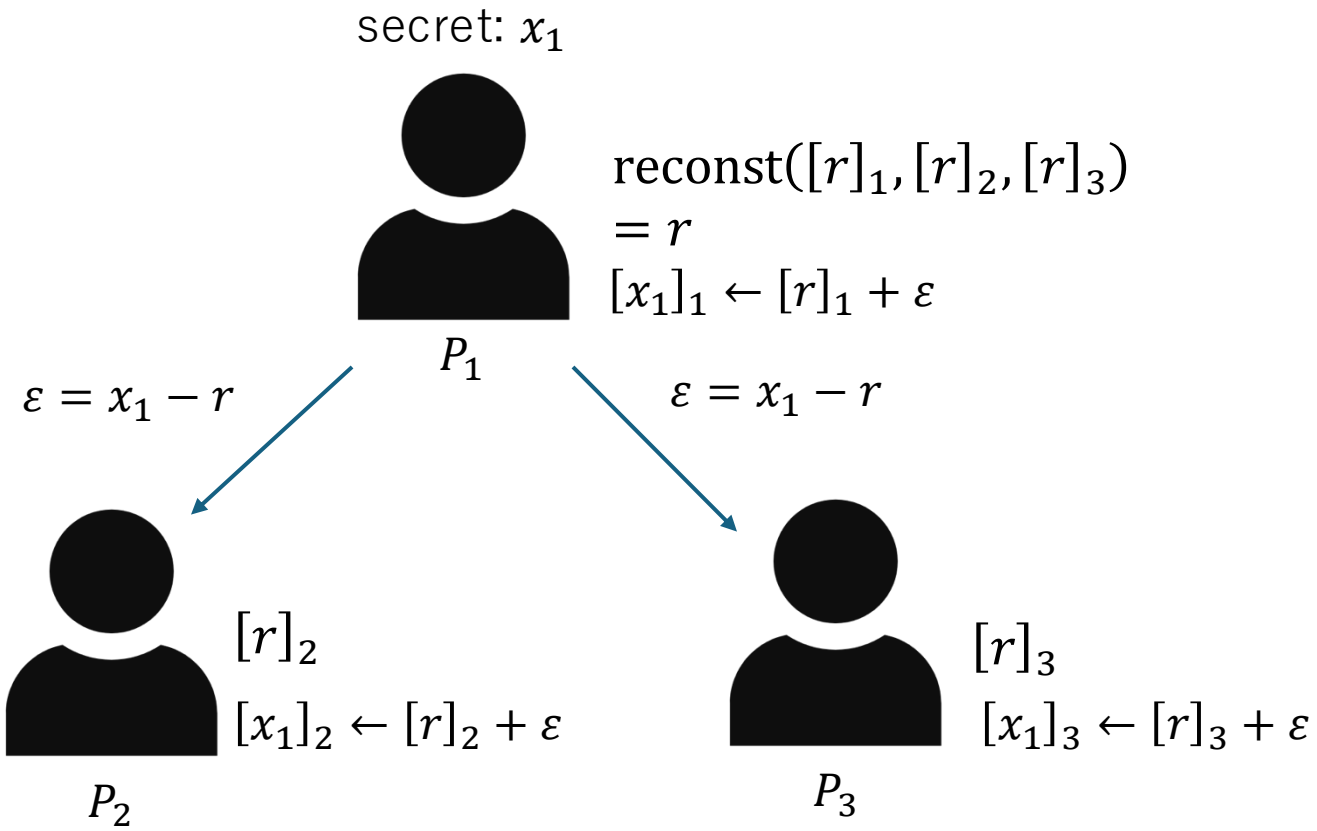


秘密分散の方法



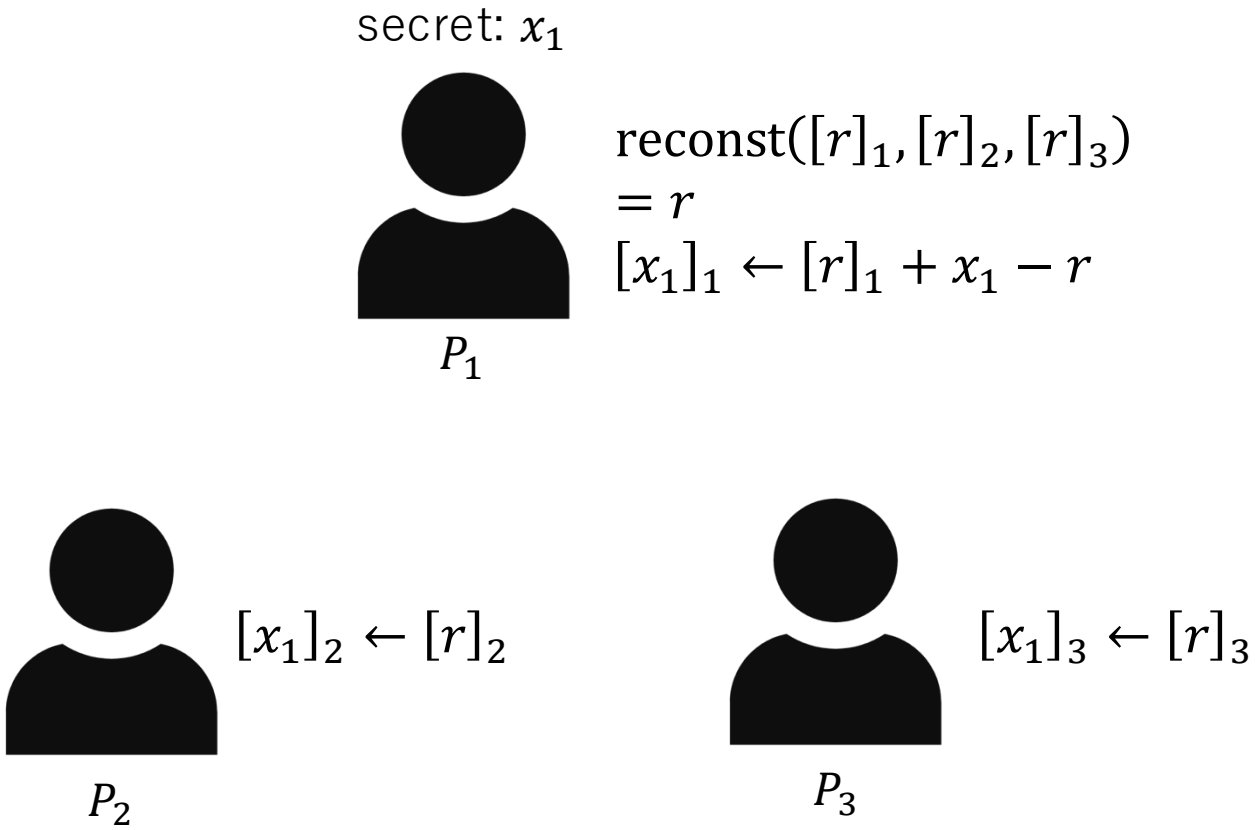
秘密分散の方法

Shamir法の場合



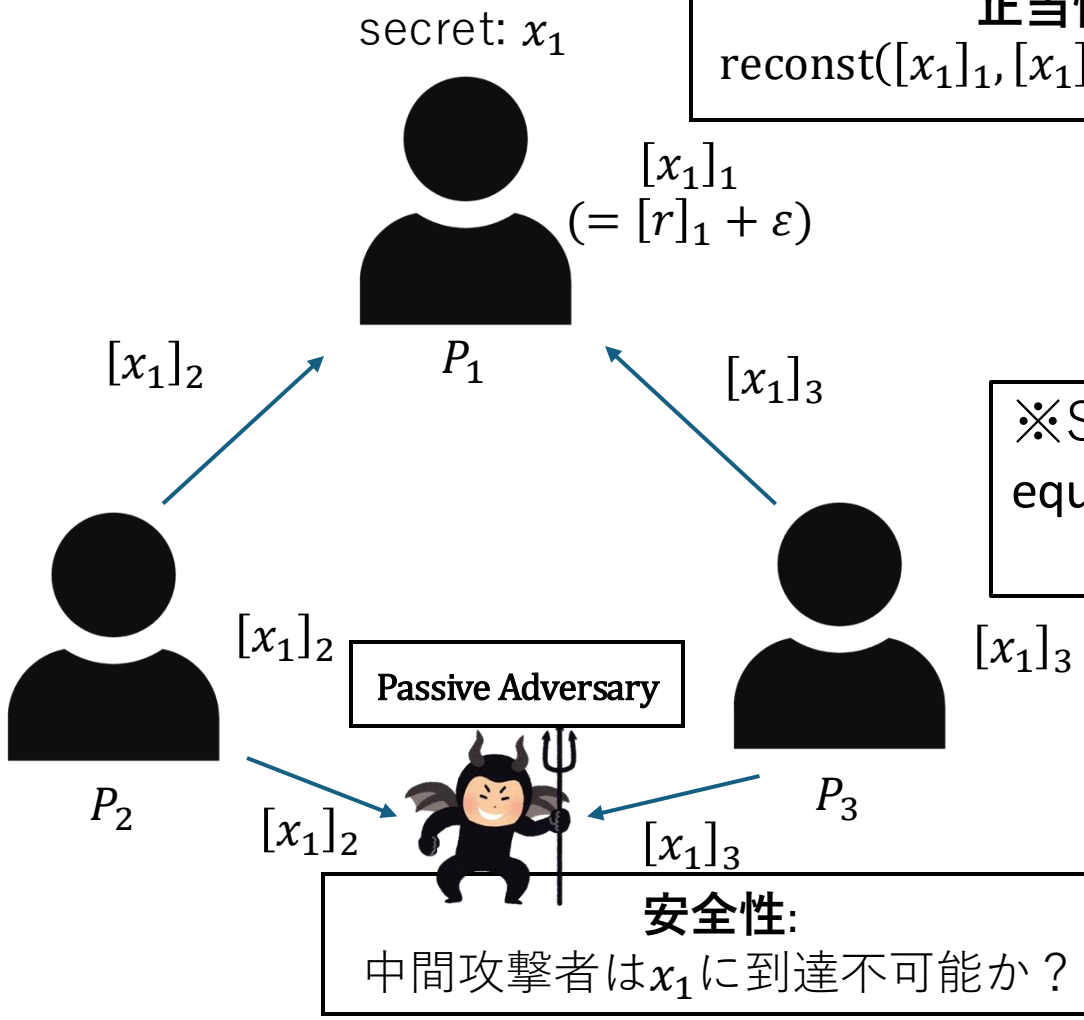
秘密分散の方法

加法の場合



秘密分散の検証

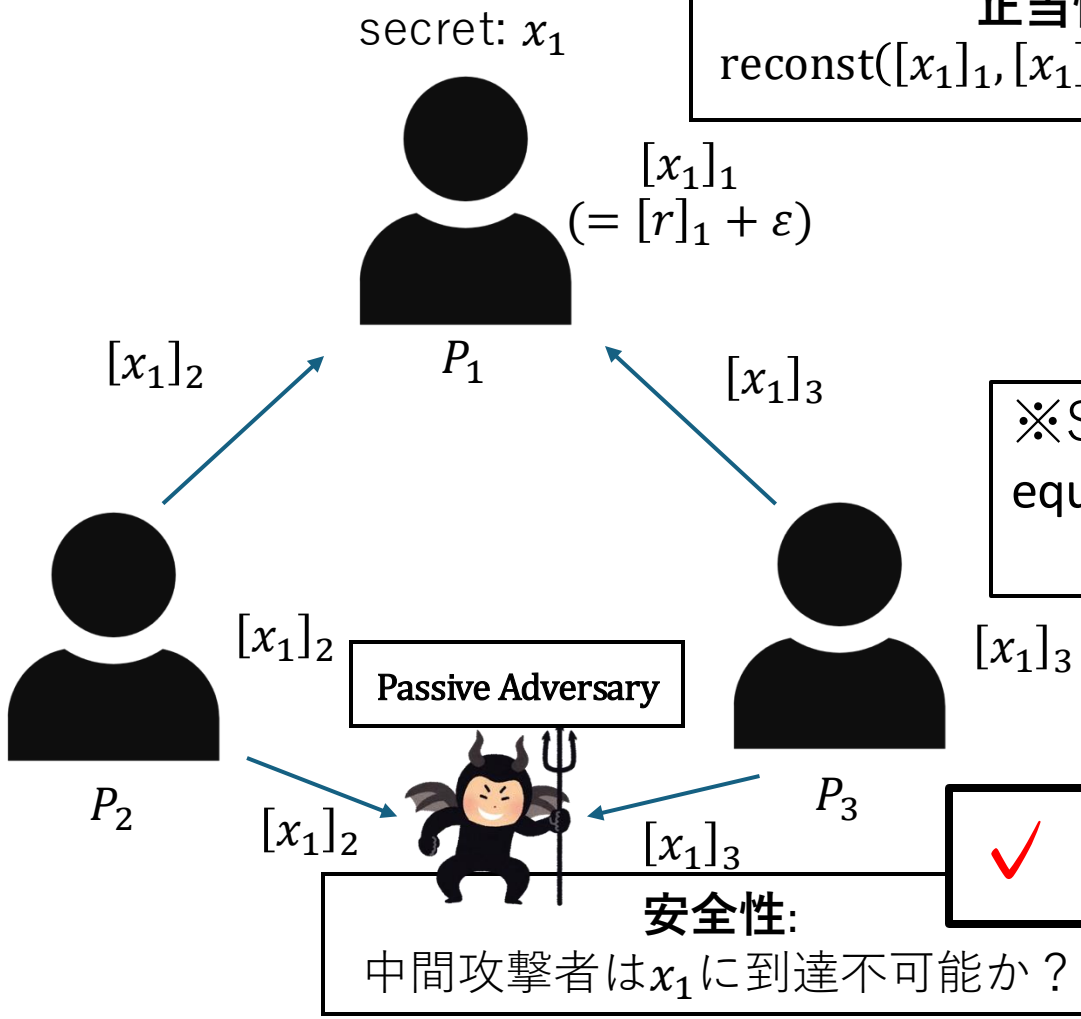
正当性:
$\mathrm{reconst}([x_1]_1, [x_1]_2, [x_1]_3) \stackrel{?}{=} x_1$



※Shamir法の場合には以下の関係性が必要
equation forall a,b:unshare;
addition(b,subtraction(a,b)) = a.

安全性:
中間攻撃者はx_1に到達不可能か？

秘密分散の検証



正当性:
$\text{reconst}([x_1]_1, [x_1]_2, [x_1]_3) ? = x_1$

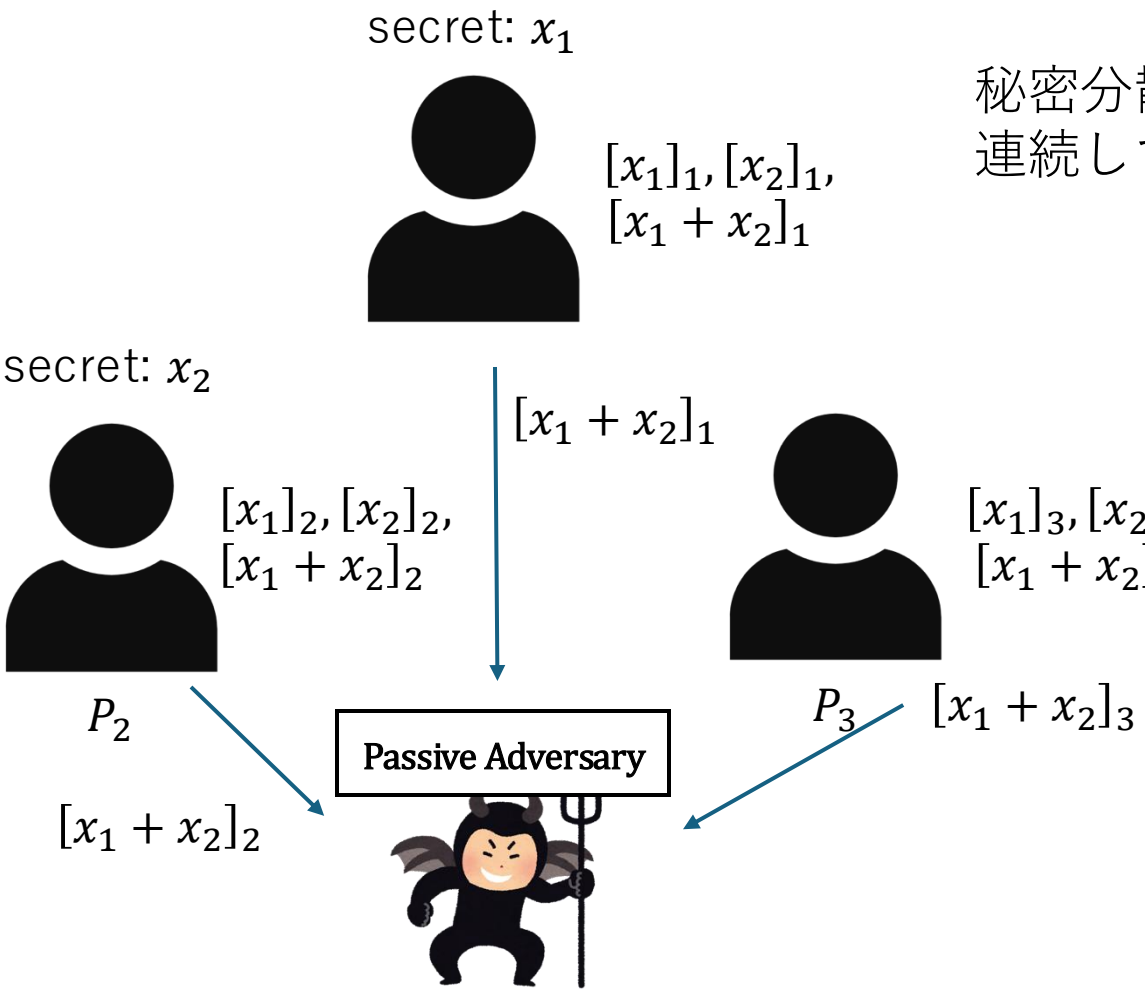
✓

※Shamir法の場合には以下の関係性が必要
equation forall a,b:unshare;
addition(b,subtraction(a,b)) = a.

✓

安全性:
中間攻撃者はx_1に到達不可能か？

シェア同士の加算の検証

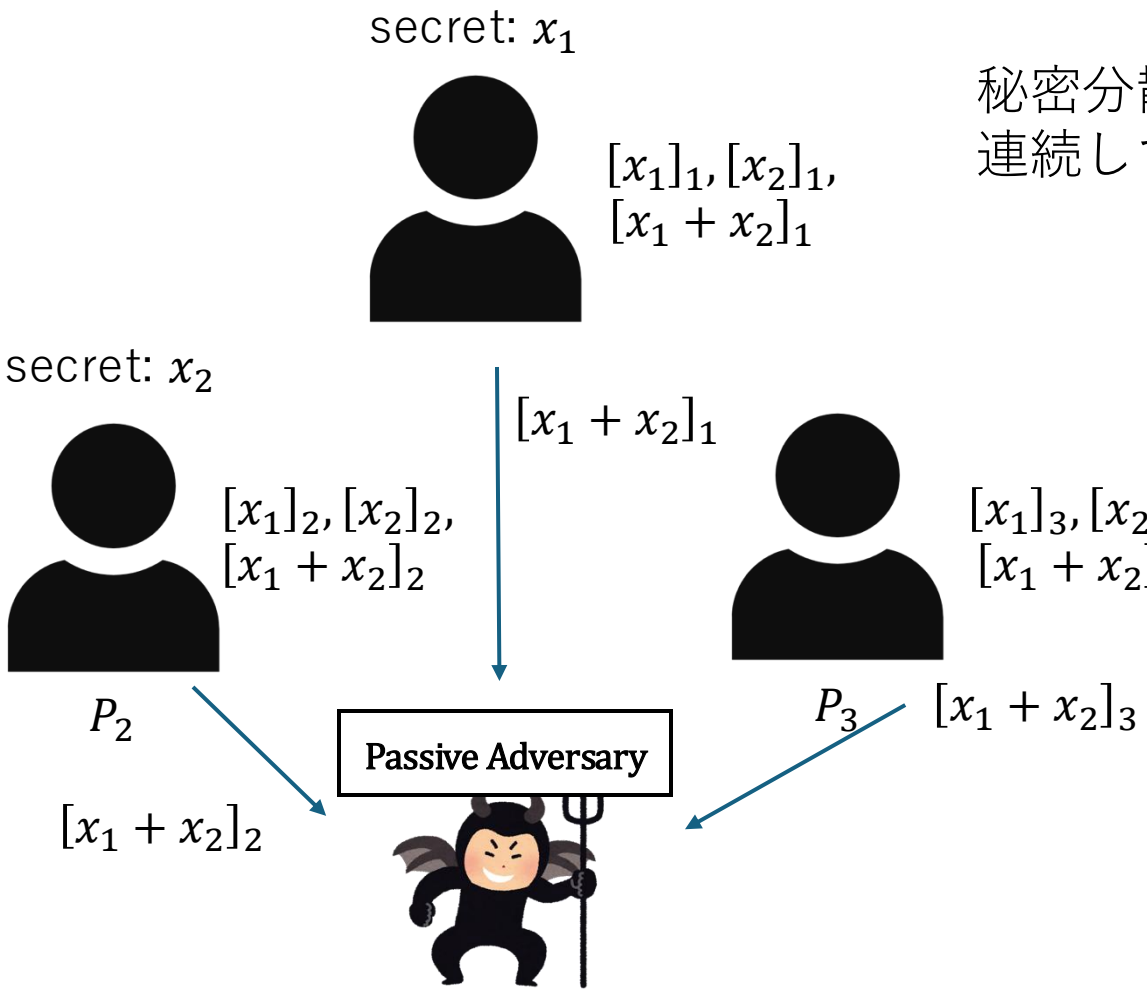


秘密分散フェーズから
連続してシェア同士の加算を行った場合の検証

正当性：
シェアを集めたら$x_1 + x_2$に到達可能か？

安全性：
中間攻撃者はx_1, x_2に到達不可能か？

シェア同士の加算の検証



秘密分散フェーズから
連続してシェア同士の加算を行った場合の検証

正当性： ✓
シェアを集めたら$x_1 + x_2$に到達可能か？

安全性： ✓
中間攻撃者はx_1, x_2に到達不可能か？

シェア同士の乗算

- $[x]_i * [y]_i = [x * y]_i$を実現するプロトコル

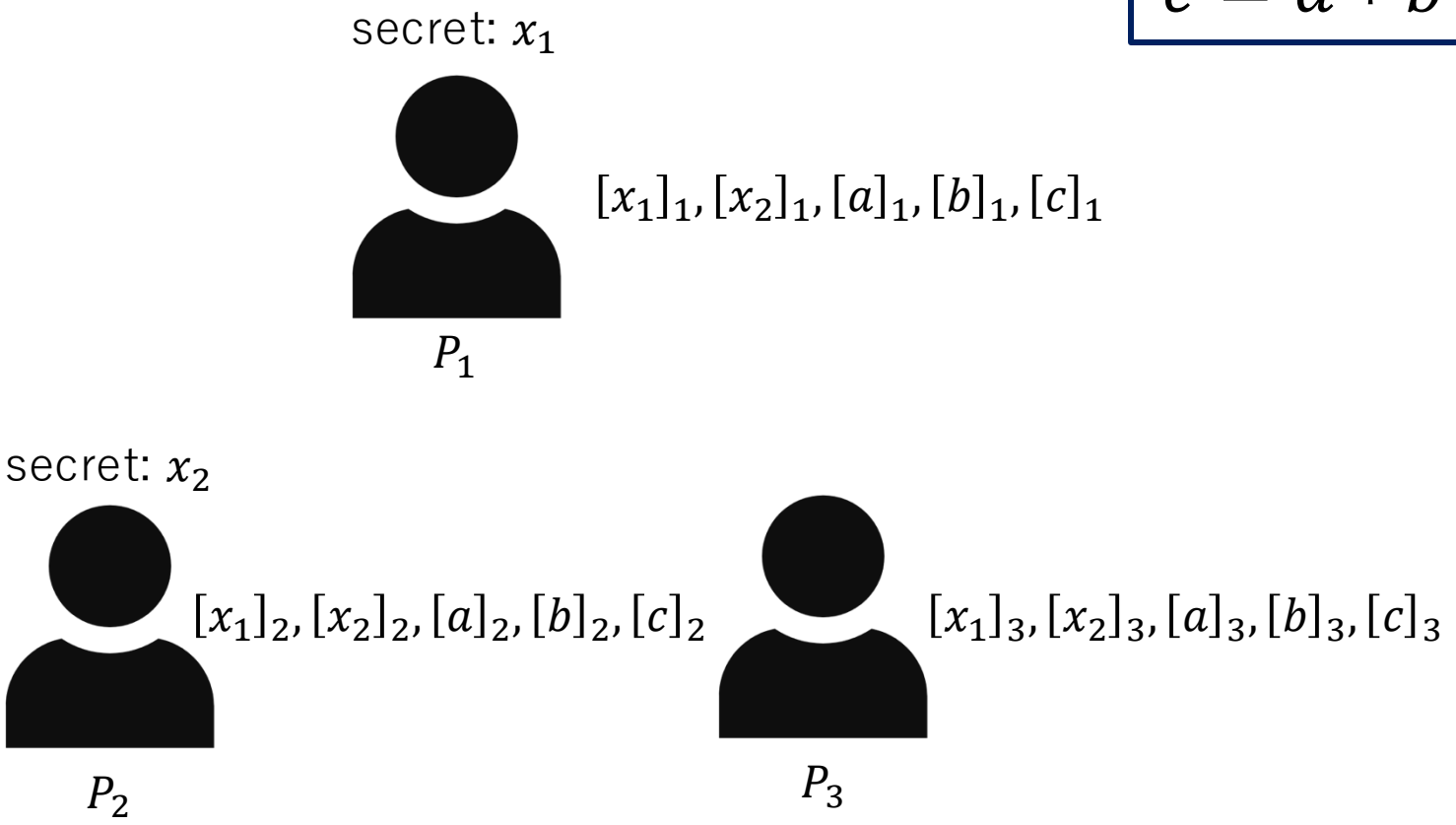
- [Bea91]では事前計算を行うことで
  オンライン通信量を削減できることが知られている

- $c = a * b$となる(入力とは無関係な)乱数のシェア$[a]_i, [b]_i, [c]_i$を
  各パーティが事前計算でシェアすることで効率的に乗算を行う

[Bea91] D. Beaver, “Efficient Multiparty Protocols Using Circuit Randomization”, CRYPTO1991.

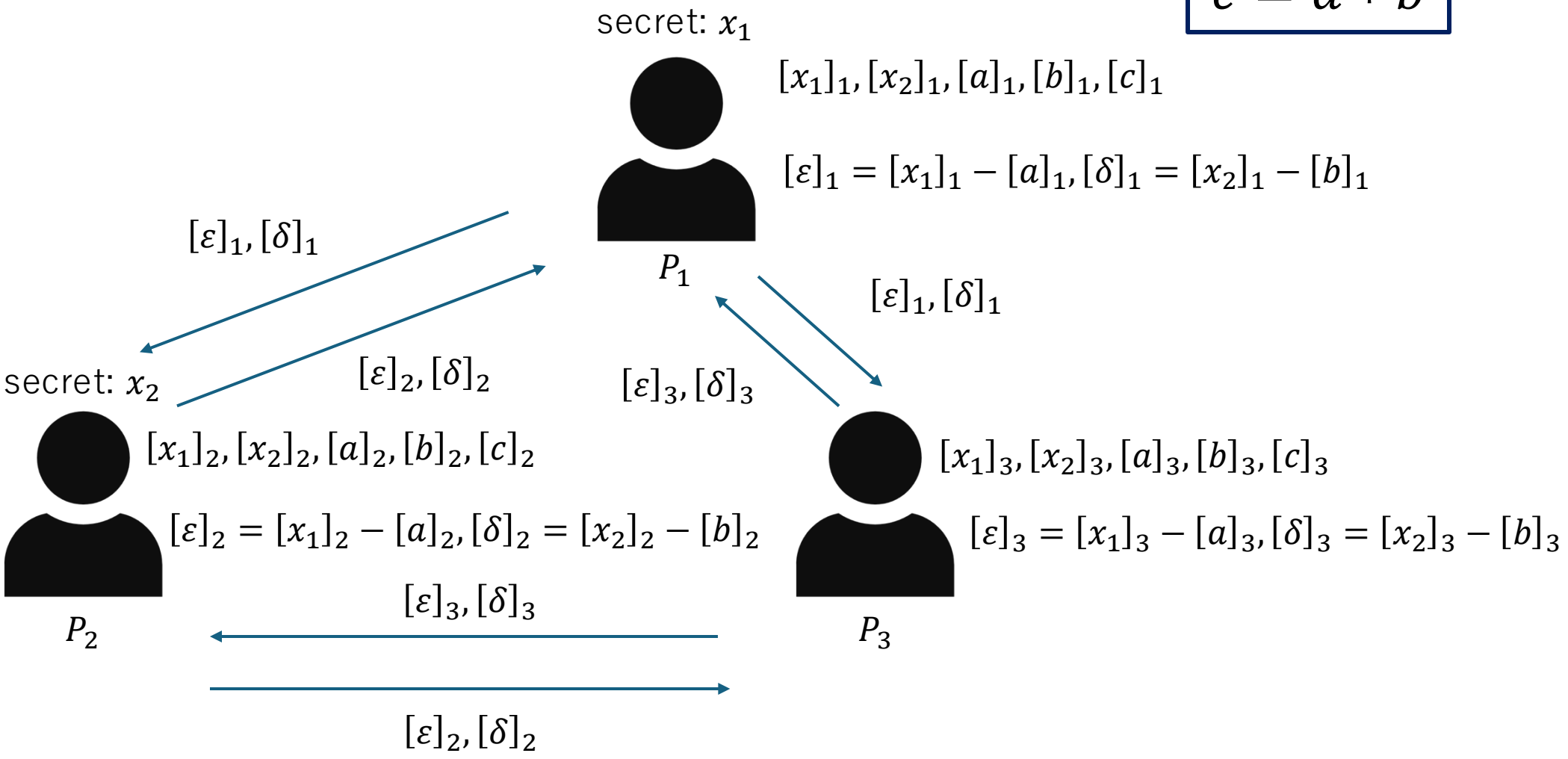
シェア同士の乗算 [Bea91]

$$c = a * b$$



シェア同士の乗算 [Bea91]

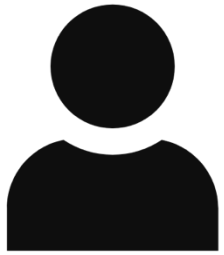
$$c = a * b$$



シェア同士の乗算 [Bea91]

$$c = a * b$$

secret: x_1



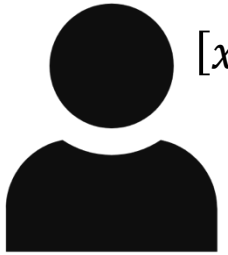
P_1

$[x_1]_1, [x_2]_1, [a]_1, [b]_1, [c]_1$

$[\varepsilon]_1 = [x_1]_1 - [a]_1, [\delta]_1 = [x_2]_1 - [b]_1$

$\varepsilon = \mathrm{reconst}([\varepsilon]_1, [\varepsilon]_2, [\varepsilon]_3),$
$\delta = \mathrm{reconst}([\delta]_1, [\delta]_2, [\delta]_3)$

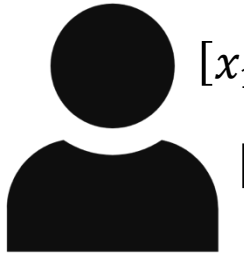
secret: x_2



P_2

$[x_1]_2, [x_2]_2, [a]_2, [b]_2, [c]_2$

$[\varepsilon]_2 = [x_1]_2 - [a]_2, [\delta]_2 = [x_2]_2 - [b]_2$
$\varepsilon = \mathrm{reconst}([\varepsilon]_1, [\varepsilon]_2, [\varepsilon]_3),$
$\delta = \mathrm{reconst}([\delta]_1, [\delta]_2, [\delta]_3)$



P_3

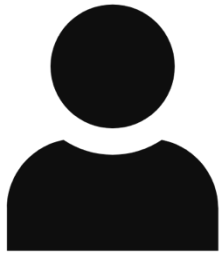
$[x_1]_3, [x_2]_3, [a]_3, [b]_3, [c]_3$

$[\varepsilon]_3 = [x_1]_3 - [a]_3, [\delta]_3 = [x_2]_3 - [b]_3$
$\varepsilon = \mathrm{reconst}([\varepsilon]_1, [\varepsilon]_2, [\varepsilon]_3),$
$\delta = \mathrm{reconst}([\delta]_1, [\delta]_2, [\delta]_3)$

シェア同士の乗算 [Bea91]

$$c = a * b$$

secret: x_1



P_1

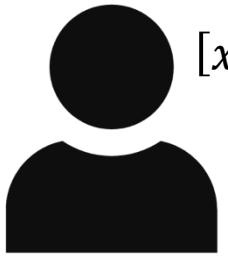
$[x_1]_1, [x_2]_1, [a]_1, [b]_1, [c]_1$

$[\varepsilon]_1 = [x_1]_1 - [a]_1, [\delta]_1 = [x_2]_1 - [b]_1$

$\varepsilon = \mathrm{reconst}([\varepsilon]_1, [\varepsilon]_2, [\varepsilon]_3),$
$\delta = \mathrm{reconst}([\delta]_1, [\delta]_2, [\delta]_3)$

$[x_1 * x_2]_1 = [c]_1 + \varepsilon[b]_1 + \delta[a]_1 + \varepsilon\delta$

secret: x_2



P_2

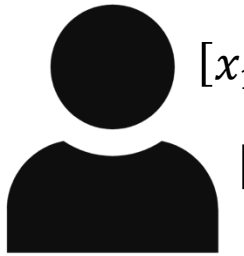
$[x_1]_2, [x_2]_2, [a]_2, [b]_2, [c]_2$

$[\varepsilon]_2 = [x_1]_2 - [a]_2, [\delta]_2 = [x_2]_2 - [b]_2$

$\varepsilon = \mathrm{reconst}([\varepsilon]_1, [\varepsilon]_2, [\varepsilon]_3),$
$\delta = \mathrm{reconst}([\delta]_1, [\delta]_2, [\delta]_3)$

$[x_1 * x_2]_2 = [c]_2 + \varepsilon[b]_2 + \delta[a]_2 + \varepsilon\delta$

(加法) $[x_1 * x_2]_2 = [c]_2 + \varepsilon[b]_2 + \delta[a]_2$



P_3

$[x_1]_3, [x_2]_3, [a]_3, [b]_3, [c]_3$

$[\varepsilon]_3 = [x_1]_3 - [a]_3, [\delta]_3 = [x_2]_3 - [b]_3$

$\varepsilon = \mathrm{reconst}([\varepsilon]_1, [\varepsilon]_2, [\varepsilon]_3),$
$\delta = \mathrm{reconst}([\delta]_1, [\delta]_2, [\delta]_3)$

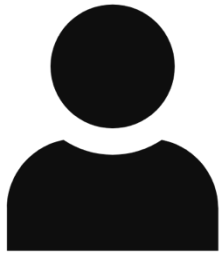
$[x_1 * x_2]_3 = [c]_3 + \varepsilon[b]_3 + \delta[a]_3 + \varepsilon\delta$

(加法) $[x_1 * x_2]_3 = [c]_3 + \varepsilon[b]_3 + \delta[a]_3$

シェア同士の乗算 [Bea91]

$$c = a * b$$

secret: x_1



P_1

$[x_1]_1, [x_2]_1, [a]_1, [b]_1, [c]_1$

$[\varepsilon]_1 = [x_1]_1 - [a]_1, [\delta]_1 = [x_2]_1 - [b]_1$

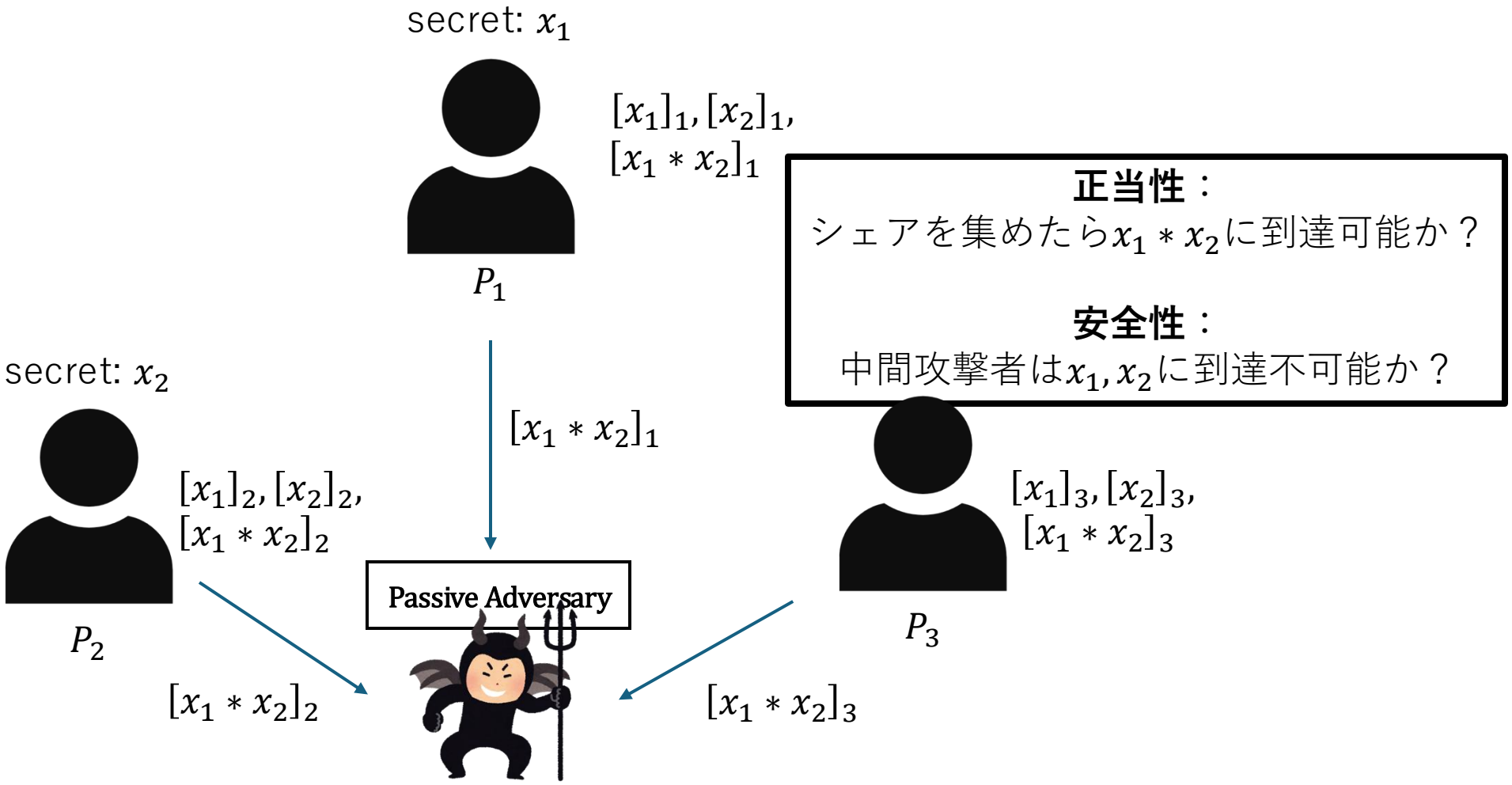
$\varepsilon = \mathrm{reconst}([\varepsilon]_1, [\varepsilon]_2, [\varepsilon]_3),$
$\delta = \mathrm{reconst}([\delta]_1, [\delta]_2, [\delta]_3)$

$[x_1 * x_2]_1 = [c]_1 + \varepsilon[b]_1 + \delta[a]_1 + \varepsilon\delta$

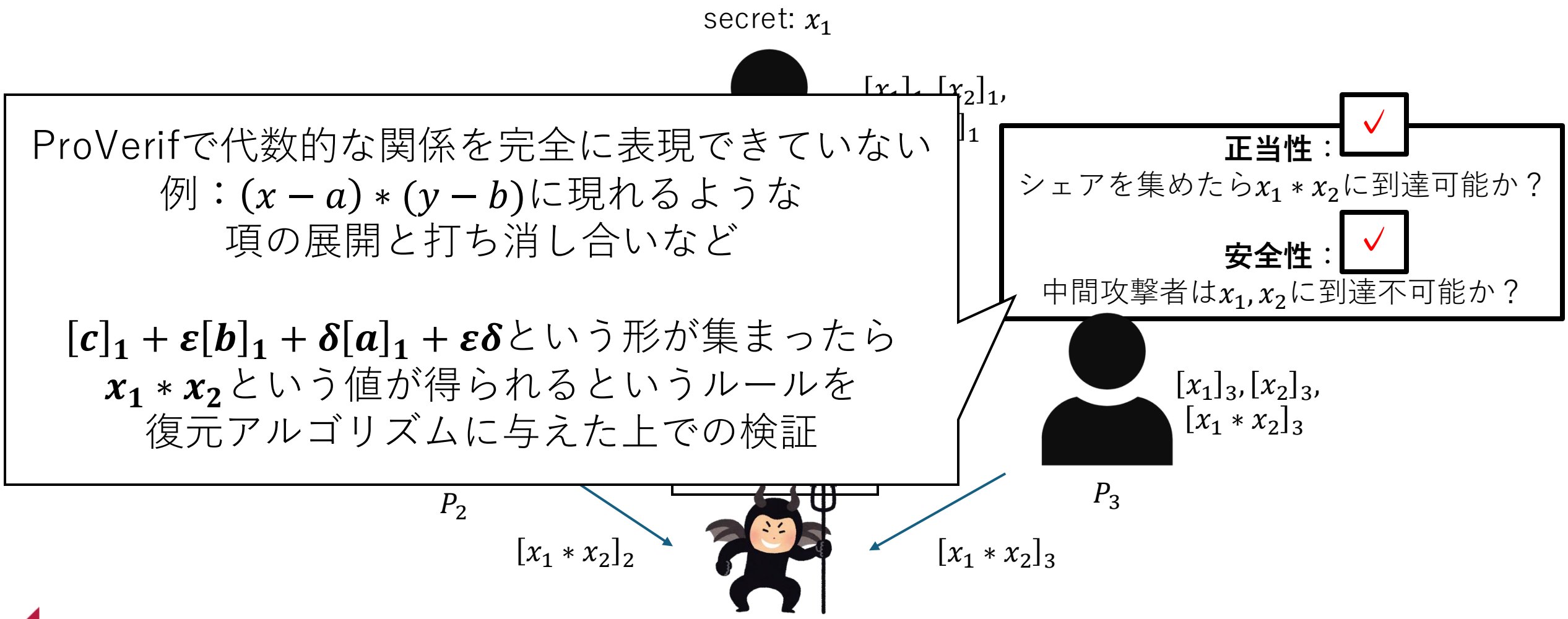
なぜこれで$x_1 * x_2$のシェアが作れているか？(正当性):

$$\mathrm{reconst}([x_1 * x_2]_1, [x_1 * x_2]_2, [x_1 * x_2]_3)$$
$$= c + \varepsilon b + \delta a + \varepsilon\delta$$
$$= ab + (x_1 - a)b + (x_2 - b)a + (x_1 - a)(x_2 - b)$$
$$= x_1 * x_2$$

シェア同士の乗算 [Bea91] の検証



シェア同士の乗算 [Bea91] の検証



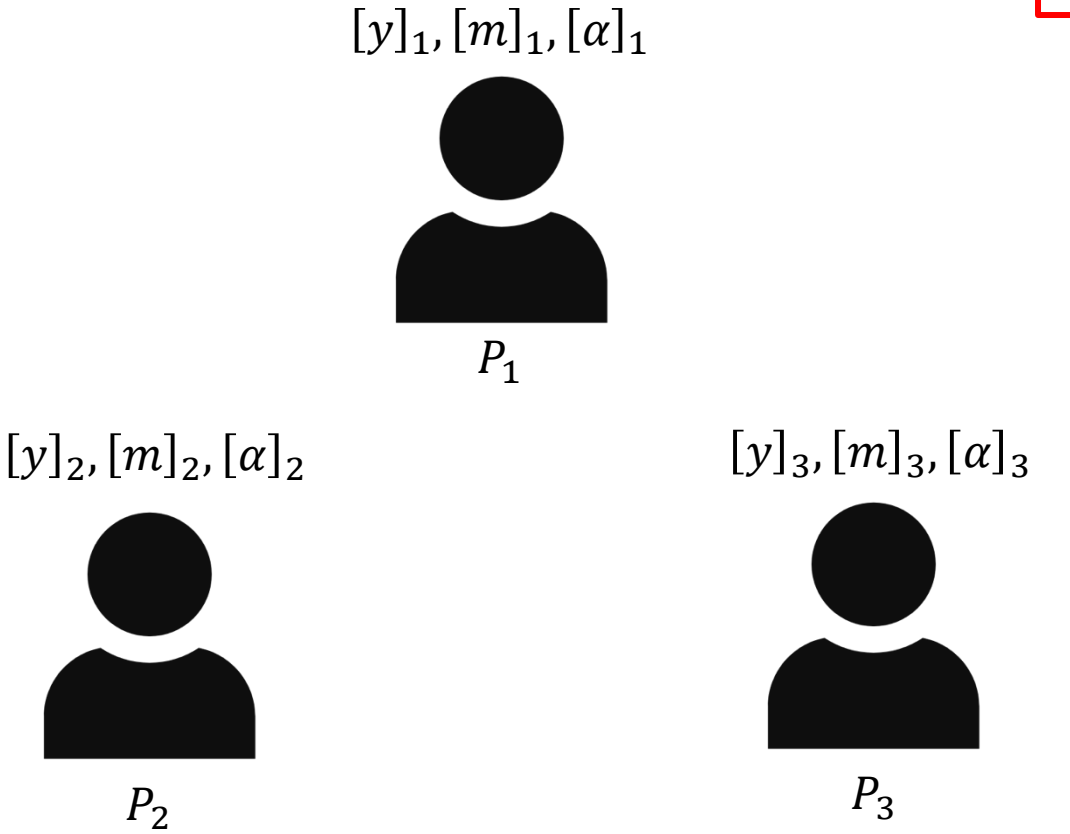
SPDZの検証について

- SPDZ：加法型秘密分散法に基づく秘密計算プロトコル
  - 前処理フェーズとオンラインフェーズからなる
- Authenticated Shareを利用して
  $n-1$人のパーティがMaliciousであっても安全性を保証する
  - Authenticated Share: $[\![x]\!]_i = (x_i, m_i, \alpha_i)$

    x_i:秘密情報xのシェア、m_i:$m(=x*\alpha)$のシェア、
    α_i:グローバルMAC鍵αのシェア
- オンラインフェーズにおける各パーティの計算後の
  計算結果の出力フェーズの安全性を検証する

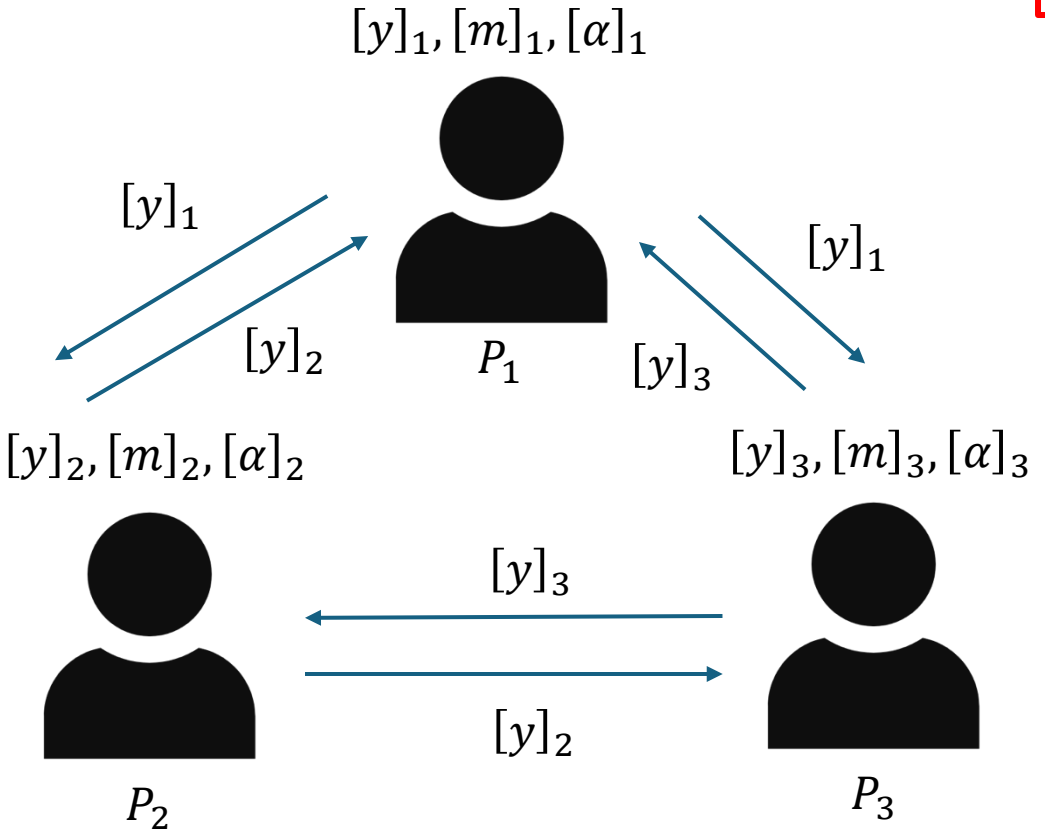
SPDZの出力フェーズ

global mac key: α
$m = mac_\alpha(y) = y * \alpha$



SPDZの出力フェーズ

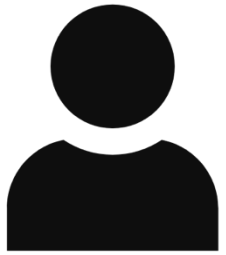
global mac key: α
$m = mac_\alpha(y) = y * \alpha$



SPDZの出力フェーズ

global mac key: $\boldsymbol{\alpha}$
$\boldsymbol{m = mac_{\alpha}(y) = y * \alpha}$

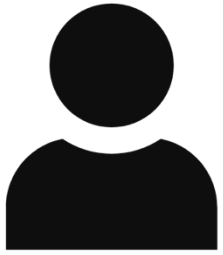
$[y]_1, [m]_1, [\alpha]_1$



$y' = \mathrm{reconst}([y]_1, [y]_2, [y]_3)$

P_1

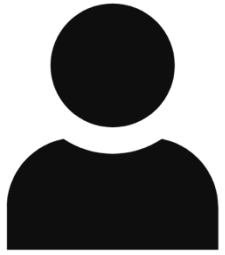
$[y]_2, [m]_2, [\alpha]_2$



$y' = \mathrm{reconst}([y]_1, [y]_2, [y]_3)$

P_2

$[y]_3, [m]_3, [\alpha]_3$

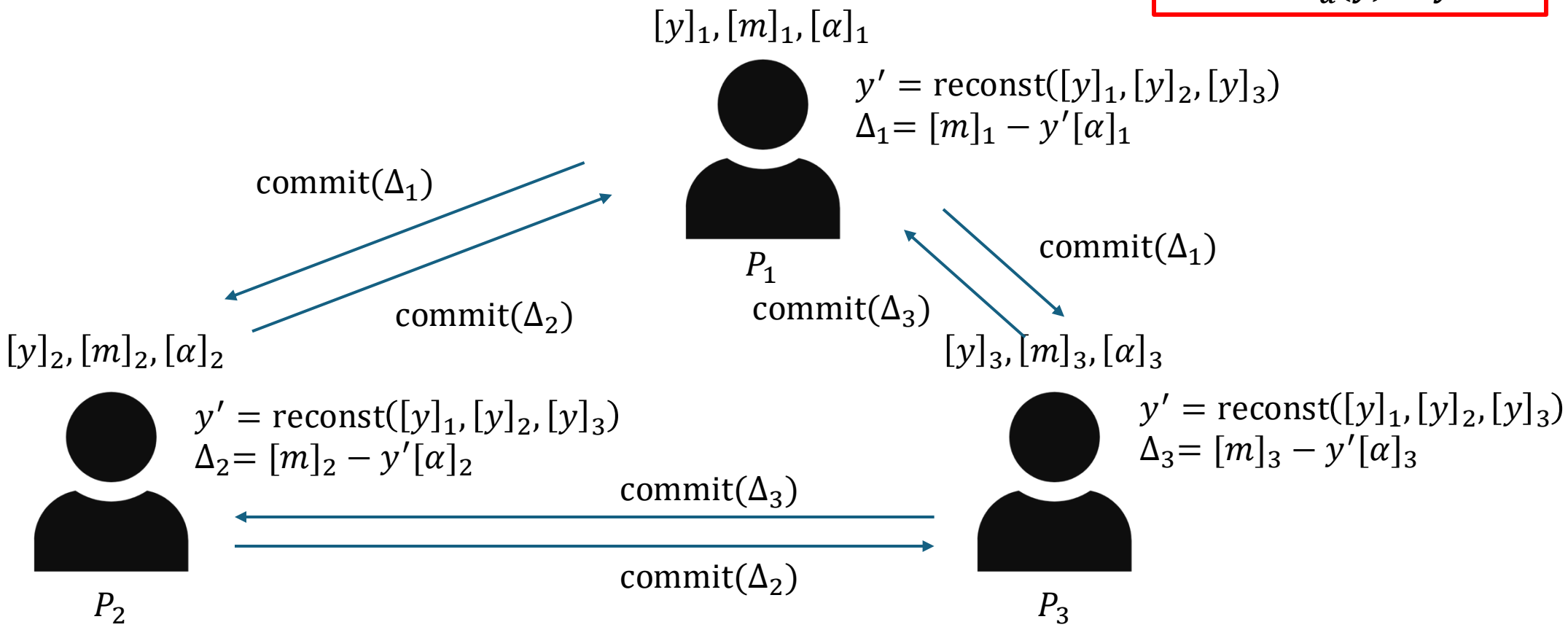


$y' = \mathrm{reconst}([y]_1, [y]_2, [y]_3)$

P_3

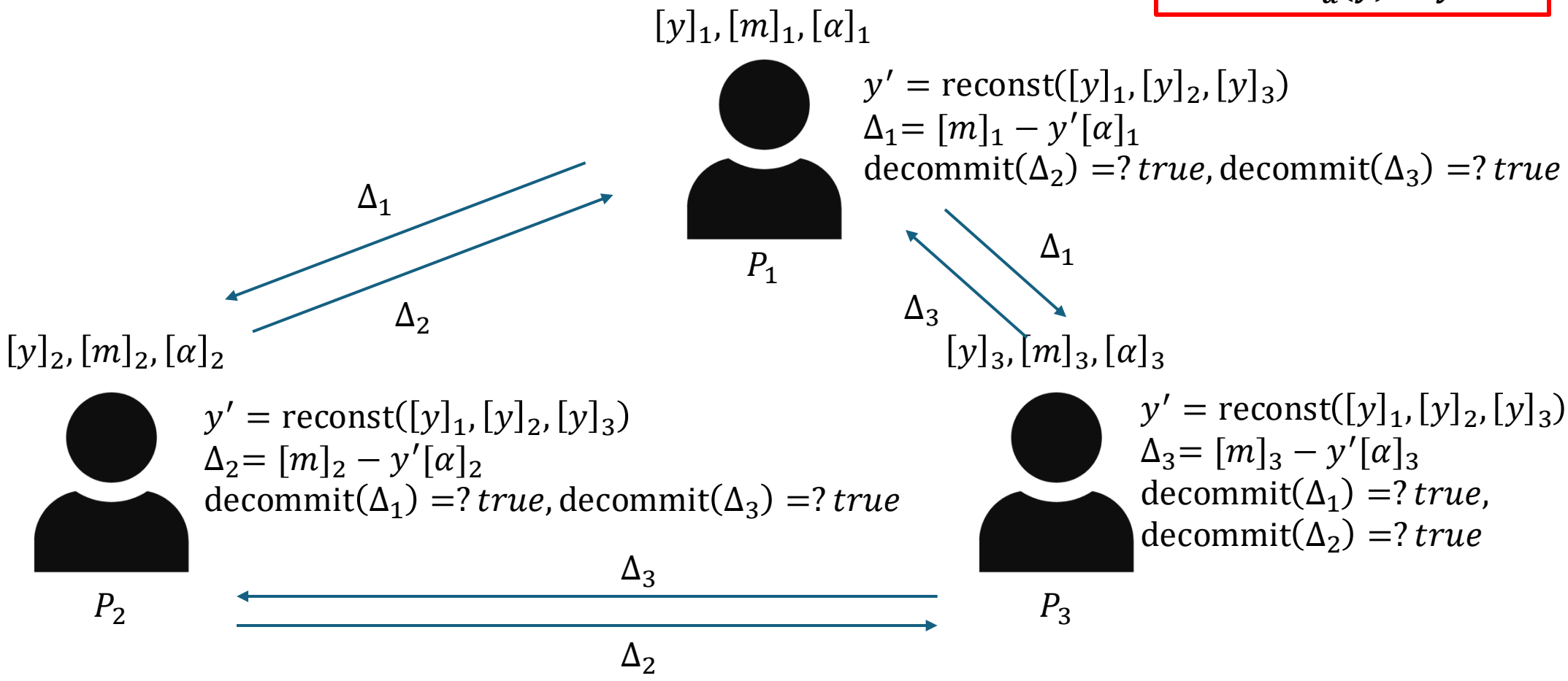
SPDZの出力フェーズ

global mac key: α
$m = mac_\alpha(y) = y * \alpha$



SPDZの出力フェーズ

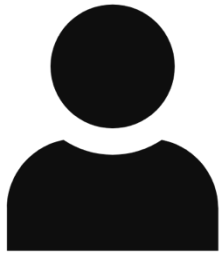
global mac key: α
$m = mac_\alpha(y) = y * \alpha$



SPDZの出力フェーズ

global mac key: $\boldsymbol{\alpha}$
$\boldsymbol{m = mac_{\alpha}(y) = y * \alpha}$

$[y]_1, [m]_1, [\alpha]_1$



$y' = \mathrm{reconst}([y]_1, [y]_2, [y]_3)$
$\Delta_1 = [m]_1 - y'[\alpha]_1$
$\mathrm{check}(\Delta_1 + \Delta_2 + \Delta_3 = 0)$

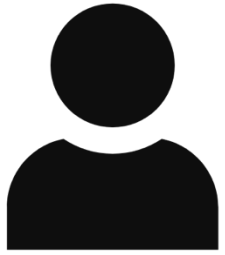
P_1

$[y]_2, [m]_2, [\alpha]_2$

$y' = \mathrm{reconst}([y]_1, [y]_2, [y]_3)$
$\Delta_2 = [m]_2 - y'[\alpha]_2$
$\mathrm{check}(\Delta_1 + \Delta_2 + \Delta_3 = 0)$

P_2

$[y]_3, [m]_3, [\alpha]_3$



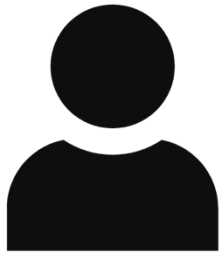
$y' = \mathrm{reconst}([y]_1, [y]_2, [y]_3)$
$\Delta_3 = [m]_3 - y'[\alpha]_3$
$\mathrm{check}(\Delta_1 + \Delta_2 + \Delta_3 = 0)$

P_3

SPDZの出力フェーズ

global mac key: $\boldsymbol{\alpha}$
$\boldsymbol{m = mac_{\alpha}(y) = y * \alpha}$

$[y]_1, [m]_1, [\alpha]_1$



P_1

$y' = \mathrm{reconst}([y]_1, [y]_2, [y]_3)$
$\Delta_1 = [m]_1 - y'[\alpha]_1$
$\mathrm{check}(\Delta_1 + \Delta_2 + \Delta_3 = 0)$

$[y]_2,$

$[y]_3)$

なぜこれでMACのチェックができているか？(正当性):

$$\Delta_1 + \Delta_2 + \Delta_3$$
$$= ([m]_1 - y[\alpha]_1) + ([m]_2 - y[\alpha]_2) + ([m]_3 - y[\alpha]_3)$$
$$= m - y\alpha$$
$$= 0$$

SPDZの出力フェーズの検証

正当性：
全パーティがMACの検証に成功しているときは、
必ず全パーティがコミットメントを送信しているときか？
安全性：
コラプトした2パーティが出力を得るときは、
必ず残りの1パーティがコミットメントを送信しているときか？

event(commitment_1(share1(x)):
各パーティがコミットメントを
送信した時に発生

event(end_1(x))):
各パーティが$\Delta_1 + \Delta_2 + \Delta_3 = 0$を
チェックしたときに発生
($\Delta_1 + \Delta_2 + \Delta_3 = 0$のチェックの際
0をequationで上手く表現できない
ため$x - x$の形になっているかを検証)

```
query x:unshare;
(event(end_1(x)) && event(end_2(x)) && event(end_3(x))) ==>
        (event(commitment_1(share1(x))) && event(commitment_2(share2(x))) &&
event(commitment_3(share3(x)))).

query x:unshare;
(event(end_2(x)) && event(end_3(x))) ==> event(commitment_1(share1(x))).
```

SPDZの出力フェーズの検証

✓ **正当性**：
全パーティがMACの検証に成功しているときは、
必ず全パーティがコミットメントを送信しているときか？

✓ **安全性**：
コラプトした2パーティが出力を得るときは、
必ず残りの1パーティがコミットメントを送信しているときか？

event(commitment_1(share1(x)):
各パーティがコミットメントを
送信した時に発生

event(end_1(x))):
各パーティが$\Delta_1 + \Delta_2 + \Delta_3 = 0$を
チェックしたときに発生
($\Delta_1 + \Delta_2 + \Delta_3 = 0$のチェックの際
0をequationで上手く表現できない
ため$x - x$の形になっているかを検証)

```
query x:unshare;
(event(end_1(x)) && event(end_2(x)) && event(end_3(x))) ==>
        (event(commitment_1(share1(x))) && event(commitment_2(share2(x))) &&
event(commitment_3(share3(x)))).

query x:unshare;
(event(end_2(x)) && event(end_3(x))) ==> event(commitment_1(share1(x))).
```

まとめ

- ProVerifによる初めての線形秘密分散の形式化
  - 秘密分散・復元アルゴリズム
  - シェア・定数による演算の線形性の形式化
  - SPDZの出力フェーズの検証への応用

- 今後の課題
  - 応用プロトコルの検証への活用