Randomized Functionalities; GMW Continued

CS 598 DH

Today's objectives

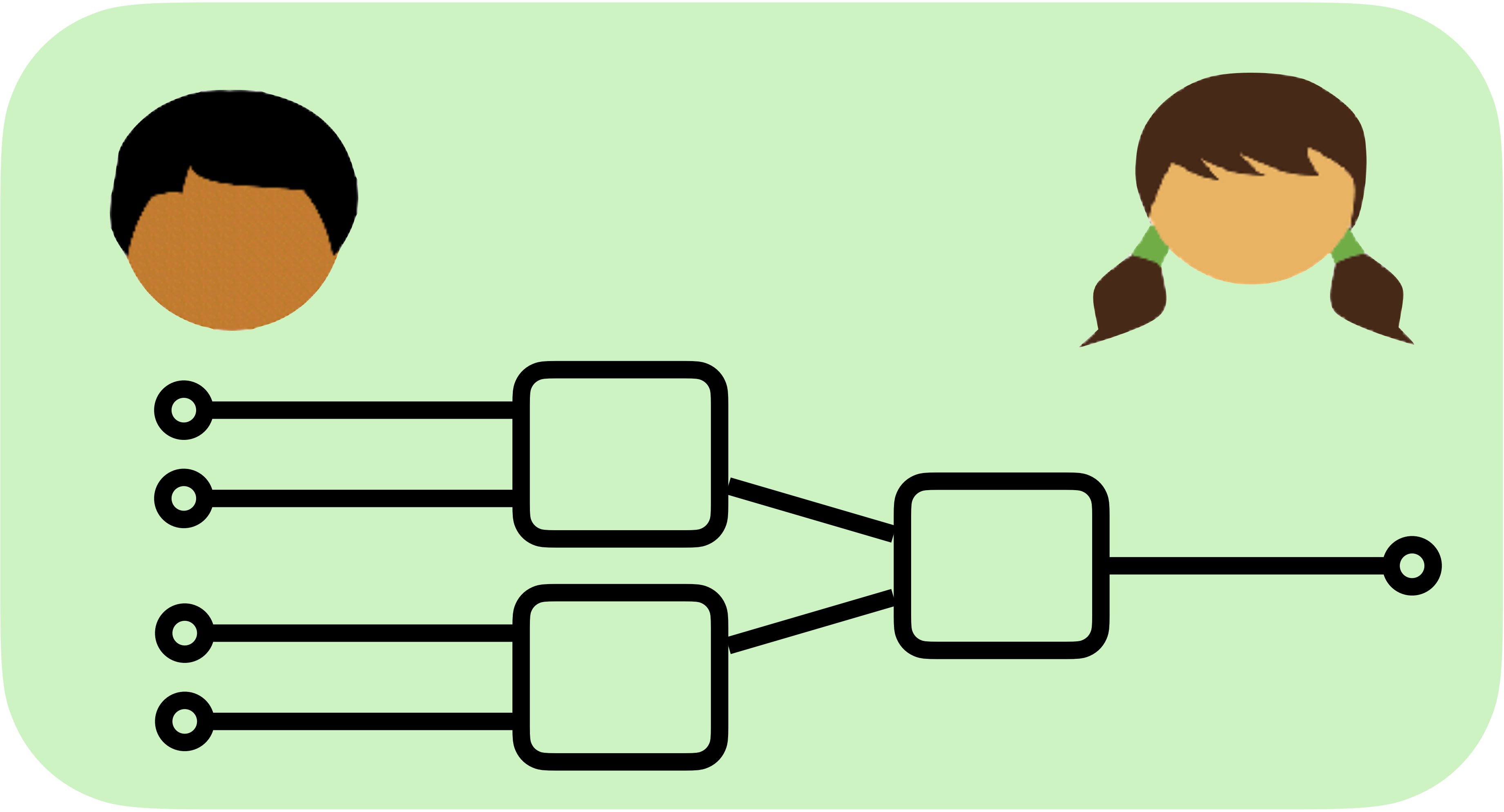
Discuss randomized functionalities

Update definition of semi-honest security

See a proof of *insecurity*

Consider security proof for GMW protocol

GMW Protocol



Propagate secret shares from input wires to output wires

Use OT to implement AND gates

Cost:

$O(|C|)$ OTs

Number of protocol rounds scales with multiplicative depth of C

Today: Full definition of semi-honest security

And GMW for more than two parties

Two-Party Semi-Honest Security
for deterministic functionalities

Let f be a function. We say that a protocol Π securely computes f in the presence of a semi-honest adversary if for each party $i \in \{0,1\}$ there exists a polynomial time simulator $\mathcal{S}_i$ *such that for all inputs x_0, x_1:*

$$\text{View}_i^{\Pi}(x_0, x_1) \overset{c}{=} \mathcal{S}_i(x_i, f(x_0, x_1))$$

Two-Party Semi-Honest Security **for deterministic functionalities**

Let f be a function. We say that a protocol Π securely computes f in the presence of a semi-honest adversary if for each party $i \in \{0,1\}$ there exists a polynomial time simulator $\mathcal{S}_i$ *such that for all inputs x_0, x_1:*

$$\text{View}_i^{\Pi}(x_0, x_1) \stackrel{c}{=} \mathcal{S}_i(x_i, f(x_0, x_1))$$

Pseudorandom Function (PRF)

A function family F is considered pseudorandom *if the following indistinguishability holds*

```
Real:
```
$k \stackrel{\$}{\leftarrow} \{0,1\}^{\lambda}$

`lookup(`m`):`

`return` $F(k, m)$

$$\stackrel{c}{=}$$

```
Ideal:
```
$T \leftarrow \text{EmptyMap}$

`lookup(`m`):`

`if` $m \notin T$`:`

$T[m] \stackrel{\$}{\leftarrow} \{0,1\}^{\text{out}}$

`return` $T[m]$

Pseudorandom Function (PRF)

A function family F is considered pseudorandom *if the following indistinguishability holds*

```
Real:
    k ←$ {0,1}^λ

    lookup(m):
        return F(k, m)
```

$$\stackrel{c}{=}$$

```
Ideal:
    T ← EmptyMap

    lookup(m):
        if m ∉ T:
            T[m] ←$ {0,1}^out
        return T[m]
```

"F looks random"

Let's "securely" implement the following functionality

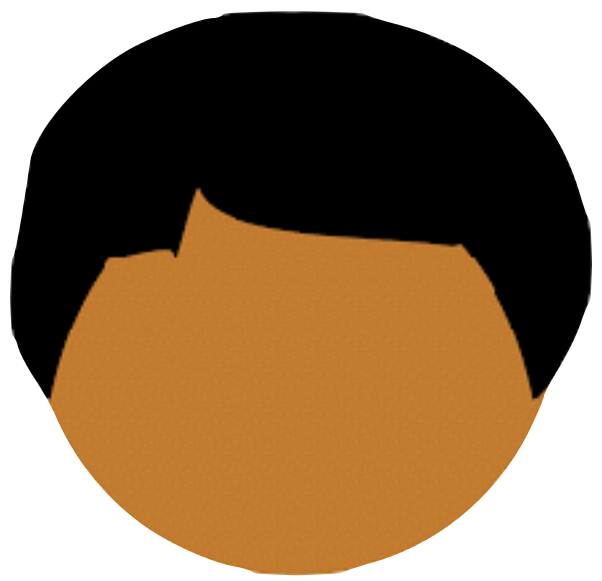
Input: P_0, P_1 input nothing

Output: P_0 outputs an encryption key k, P_1 outputs $F(k,0)$

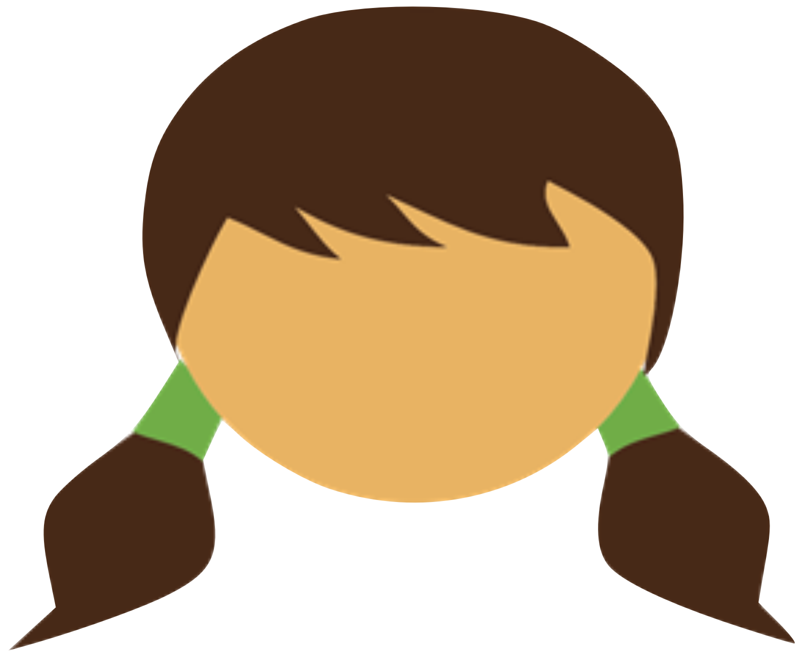
Let's "securely" implement the following functionality

Input: P_0, P_1 input nothing

Output: P_0 outputs an encryption key k, P_1 outputs $F(k,0)$



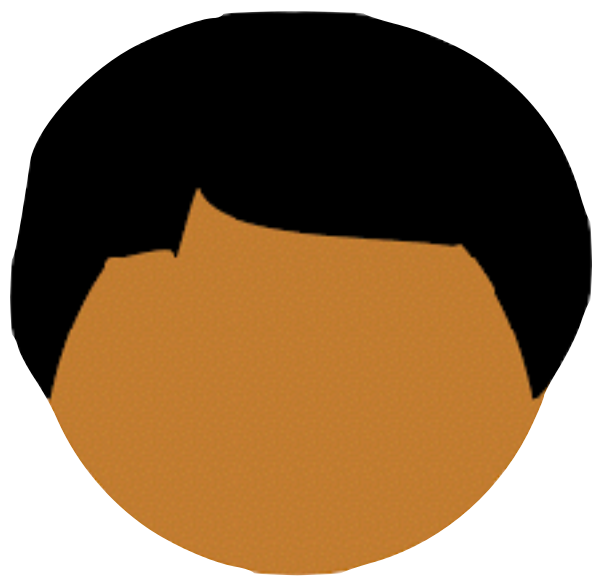
$k \xleftarrow{\$} \{0,1\}^\lambda$



Let’s “securely” implement the following functionality

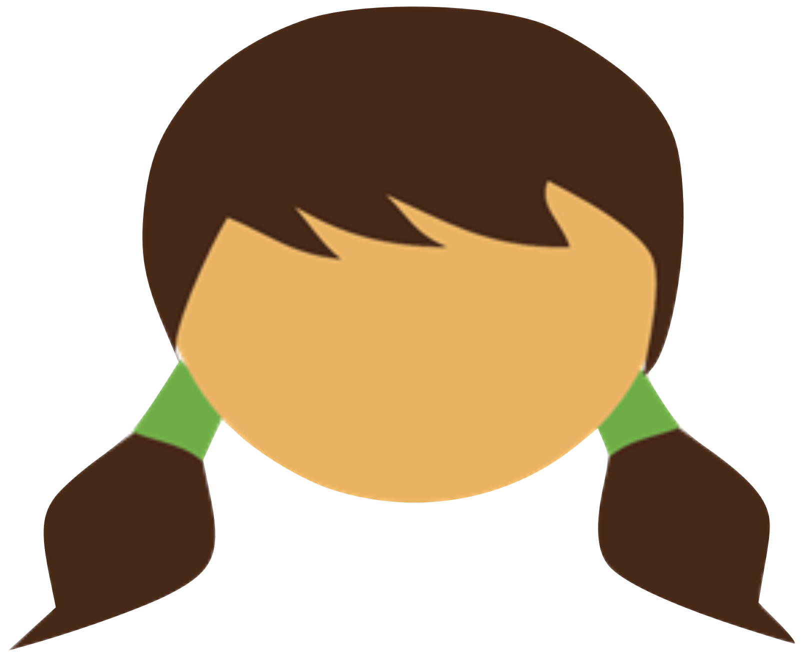
Input: P_0, P_1 input nothing

Output: P_0 outputs an encryption key k, P_1 outputs $F(k,0)$

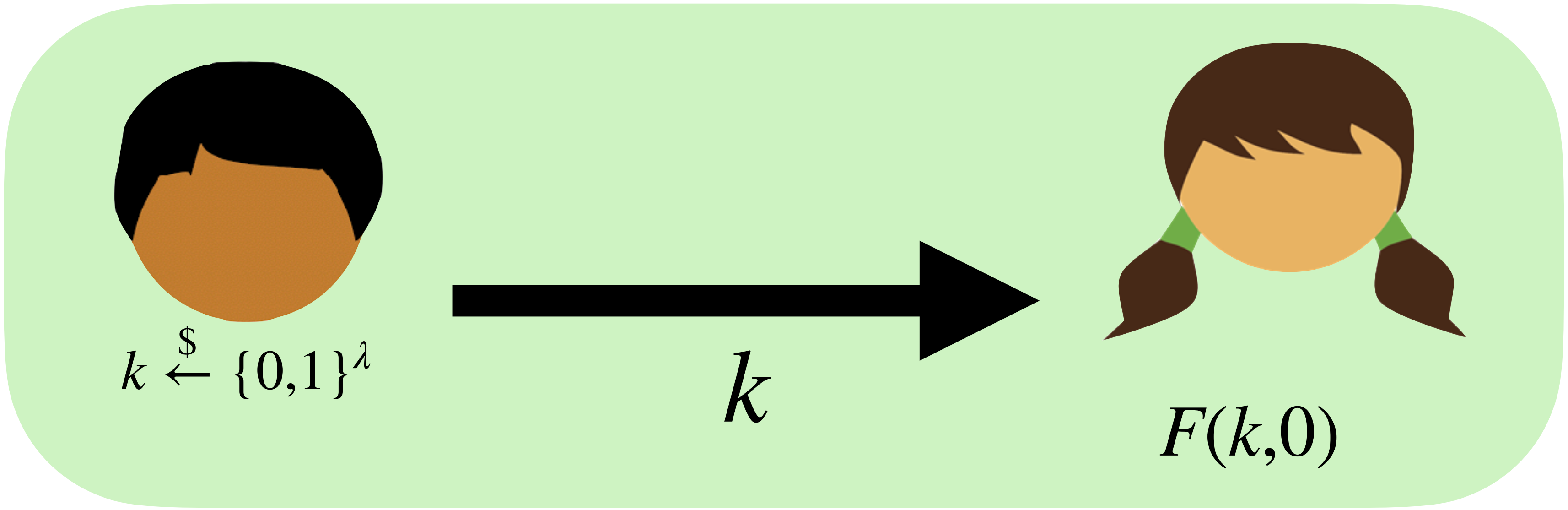


$k \xleftarrow{\$} \{0,1\}^\lambda$

$\xrightarrow{\quad k \quad}$

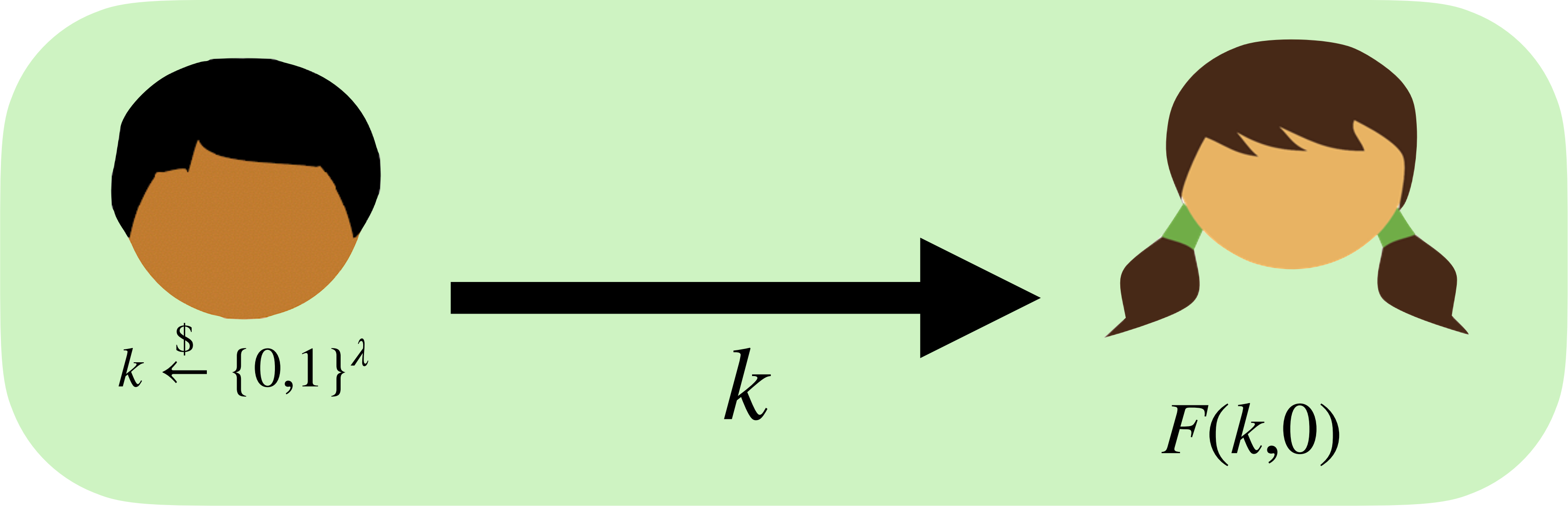


$F(k,0)$



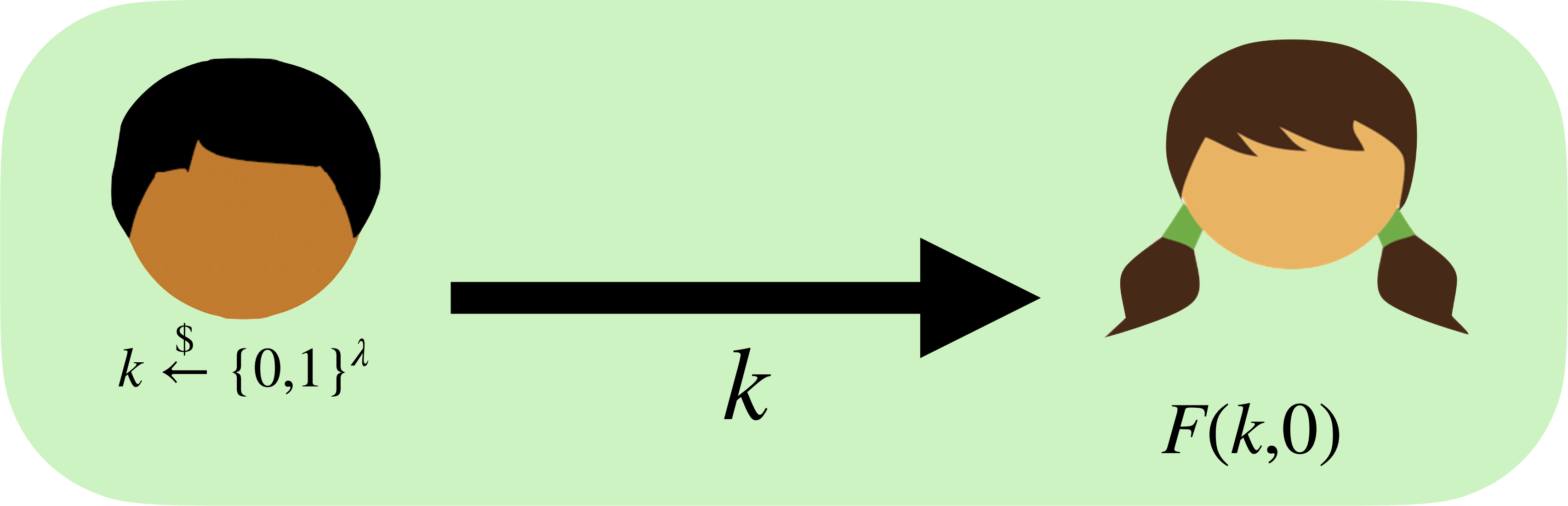
$\text{View}_0(\ ):$

$\text{View}_1(\ ):$



View$_0$( ):
$k \xleftarrow{\$} \{0,1\}^\lambda$
`return` k

View$_1$( ):
$k \xleftarrow{\$} \{0,1\}^\lambda$
`return` k



$\text{View}_0(\,)$:

$k \xleftarrow{\$} \{0,1\}^\lambda$

`return` k

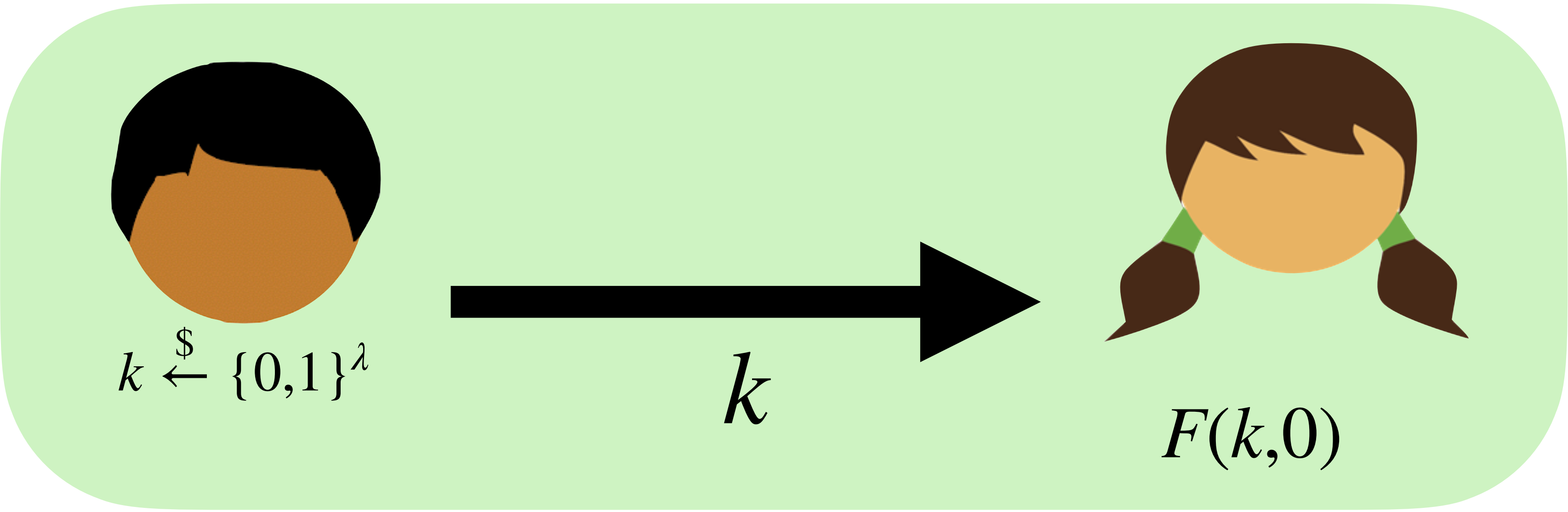
$\text{View}_1(\,)$:

$k \xleftarrow{\$} \{0,1\}^\lambda$

`return` k

$\mathcal{S}_0(k)$:

$\mathcal{S}_1(F(k,0))$:

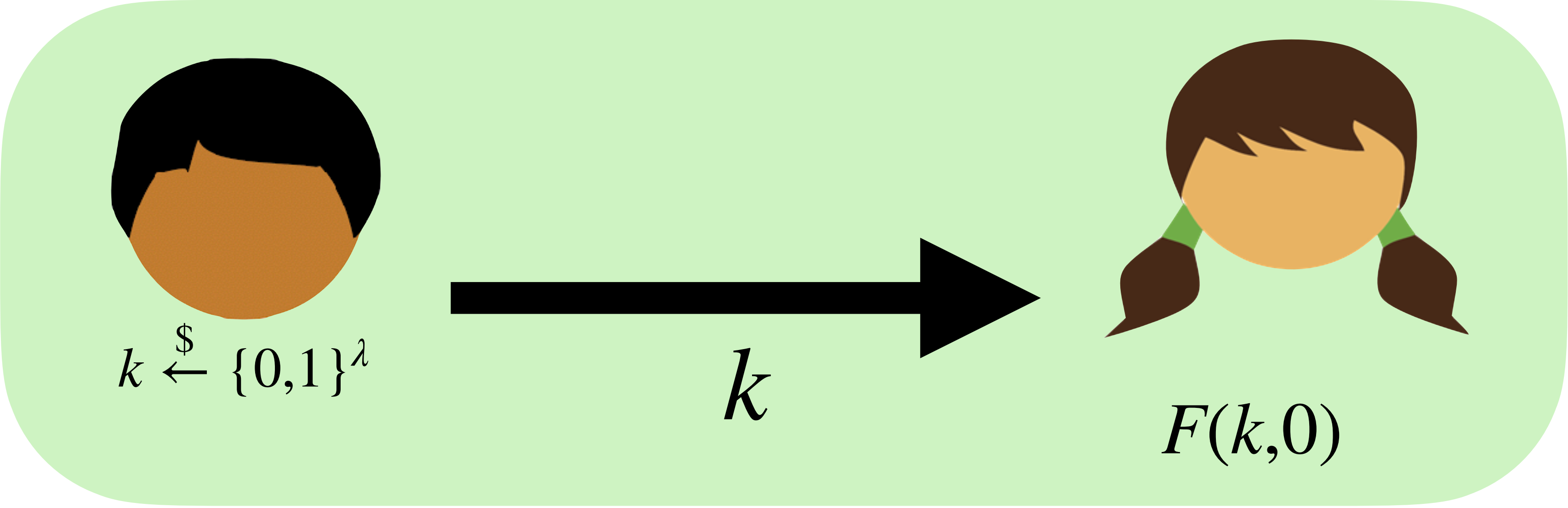


View$_0$( ):
  $k \xleftarrow{\$} \{0,1\}^\lambda$
  `return` k

View$_1$( ):
  $k \xleftarrow{\$} \{0,1\}^\lambda$
  `return` k

$\mathcal{S}_0(k)$:
  $k' \xleftarrow{\$} \{0,1\}^\lambda$
  `return` k'

$\mathcal{S}_1(F(k,0))$:



$\text{View}_0(\ )$:
  $k \xleftarrow{\$} \{0,1\}^\lambda$
  return k

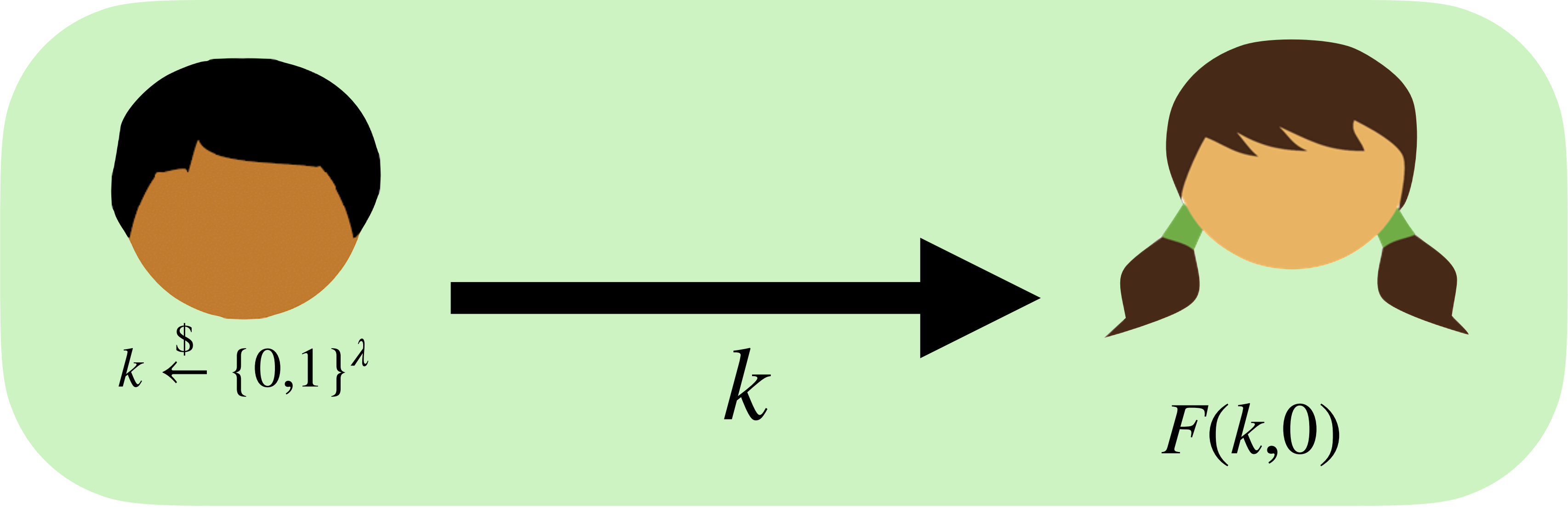
$=$

$\mathcal{S}_0(k)$:
  $k' \xleftarrow{\$} \{0,1\}^\lambda$
  return k'

$\text{View}_1(\ )$:
  $k \xleftarrow{\$} \{0,1\}^\lambda$
  return k

$=$

$\mathcal{S}_1(F(k,0))$:
  $k' \xleftarrow{\$} \{0,1\}^\lambda$
  return k'



$\text{View}_0(\ )$:

$k \xleftarrow{\$} \{0,1\}^\lambda$

`return` k

$=$

$\mathcal{S}_0(k)$:

$k' \xleftarrow{\$} \{0,1\}^\lambda$

`return` k'

The simulated view is not consistent with the output!

$\text{View}_1(\ )$:

$k \xleftarrow{\$} \{0,1\}^\lambda$

`return` k

$=$

$\mathcal{S}_1(F(k,0))$:

$k' \xleftarrow{\$} \{0,1\}^\lambda$

`return` k'

Two-Party Semi-Honest Security **for deterministic functionalities**

*Let f be a **deterministic** functionality. We say that a protocol Π securely computes f in the presence of a semi-honest adversary if for each party $i \in \{0,1\}$ there exists a polynomial time* simulator $\mathcal{S}_i$ *such that for all inputs x_0, x_1:*

$$\{\text{View}_i^\Pi(x_0, x_1)\}$$

$$\stackrel{c}{=}$$

$$\{\mathcal{S}_i(x_i, y_i) \mid (y_0, y_1) \leftarrow f(x_0, x_1)\}$$

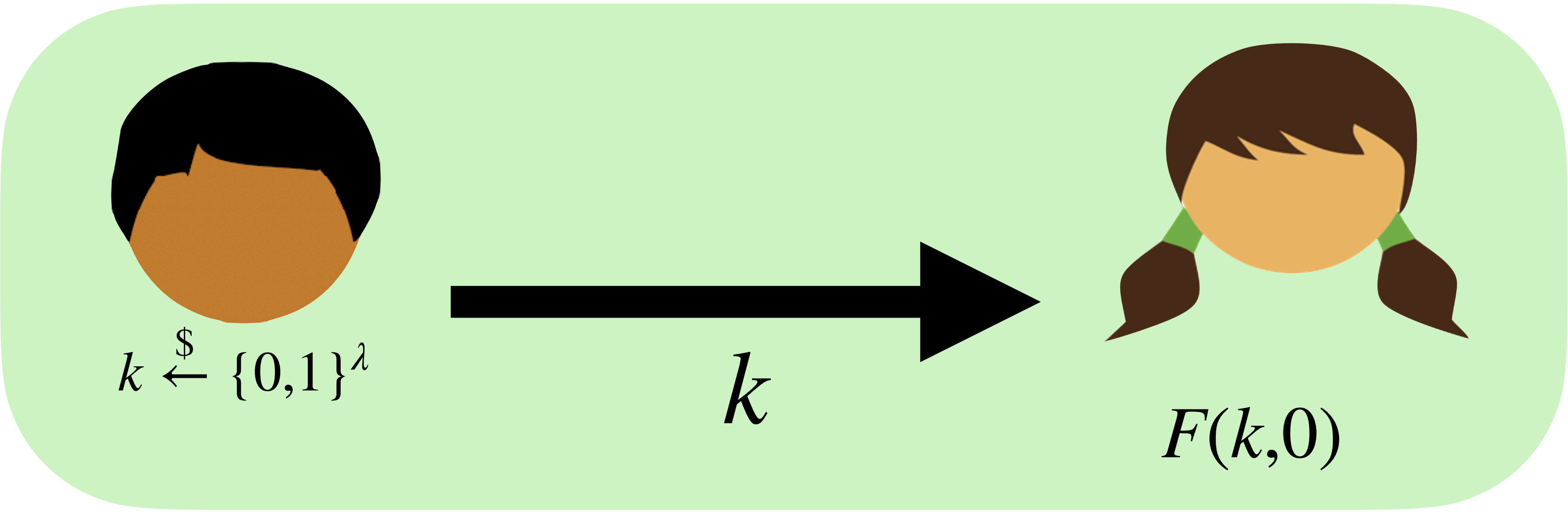
Two-Party Semi-Honest Security

Let f be a functionality. We say that a protocol Π securely computes f in the presence of a semi-honest adversary if for each party $i \in \{0,1\}$ there exists a polynomial time simulator $\mathcal{S}_i$ *such that for all inputs x_0, x_1:*

$$\{\text{View}_i^{\Pi}(x_0, x_1), \text{Output}^{\Pi}(x_0, x_1)\}$$

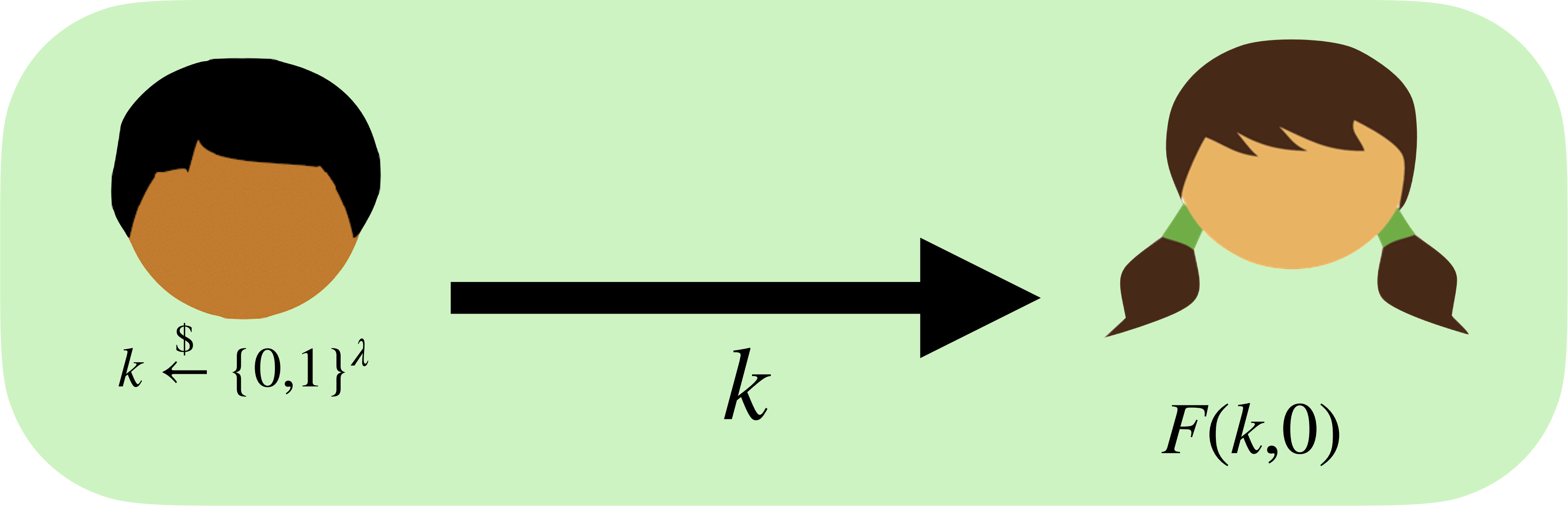
$$\stackrel{c}{=}$$

$$\{\mathcal{S}_i(x_i, y_i), (y_0, y_1) \mid (y_0, y_1) \leftarrow f(x_0, x_1)\}$$



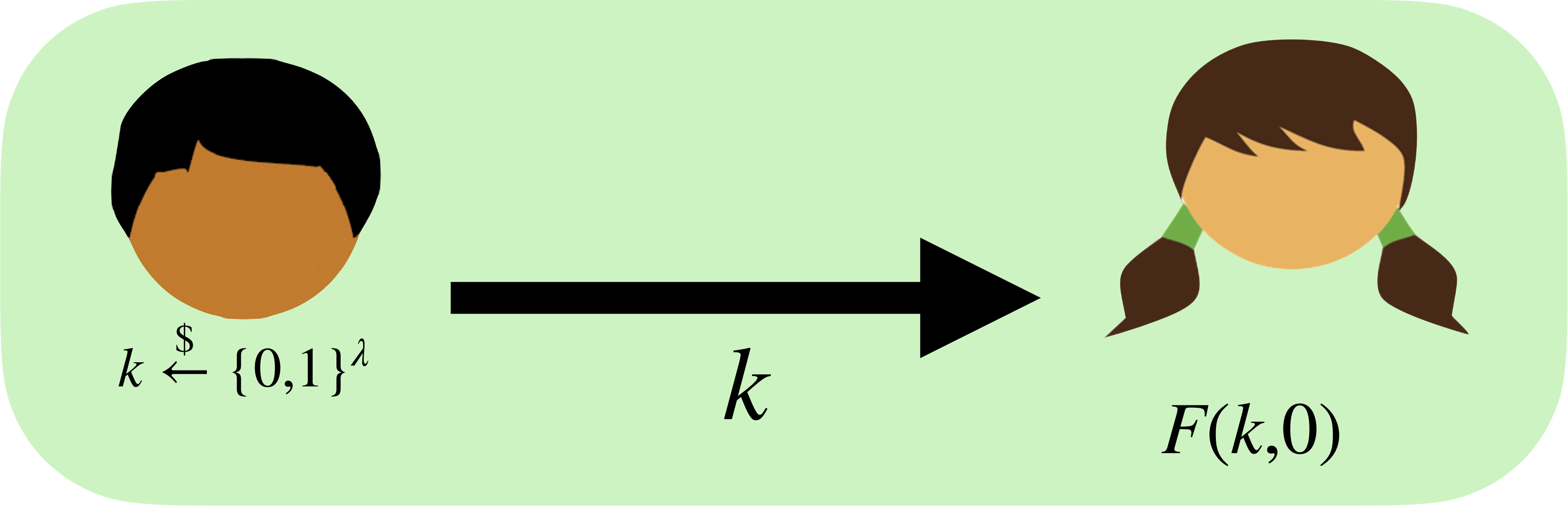
$$\{\text{View}_i^{\Pi}(x_0, x_1), \text{Output}^{\Pi}(x_0, x_1)\} \stackrel{c}{=} \{\mathcal{S}_i(x_i, y_i), (y_0, y_1) \mid (y_0, y_1) \leftarrow f(x_0, x_1)\}$$

$$\{k, (k, F(k,0))\} \stackrel{c}{=} \{\mathcal{S}_1(F(k,0)), (k, F(k,0)) \mid k \leftarrow \{0,1\}^{\lambda}\}$$



Fact: there does not exist $\mathcal{S}_1$ proving this protocol secure

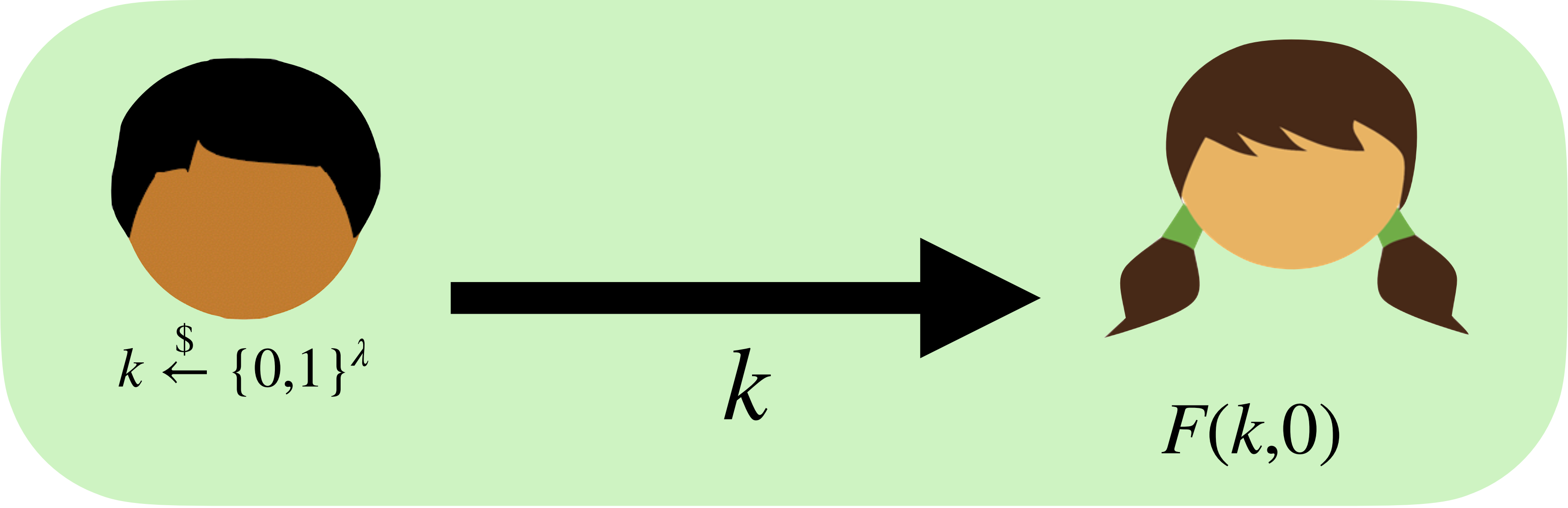
$$\{k, (k, F(k,0))\} \overset{c}{=} \{\mathcal{S}_1(F(k,0)), (k, F(k,0)) \mid k \leftarrow \{0,1\}^\lambda\}$$



Fact: there does not exist $\mathcal{S}_1$ proving this protocol secure

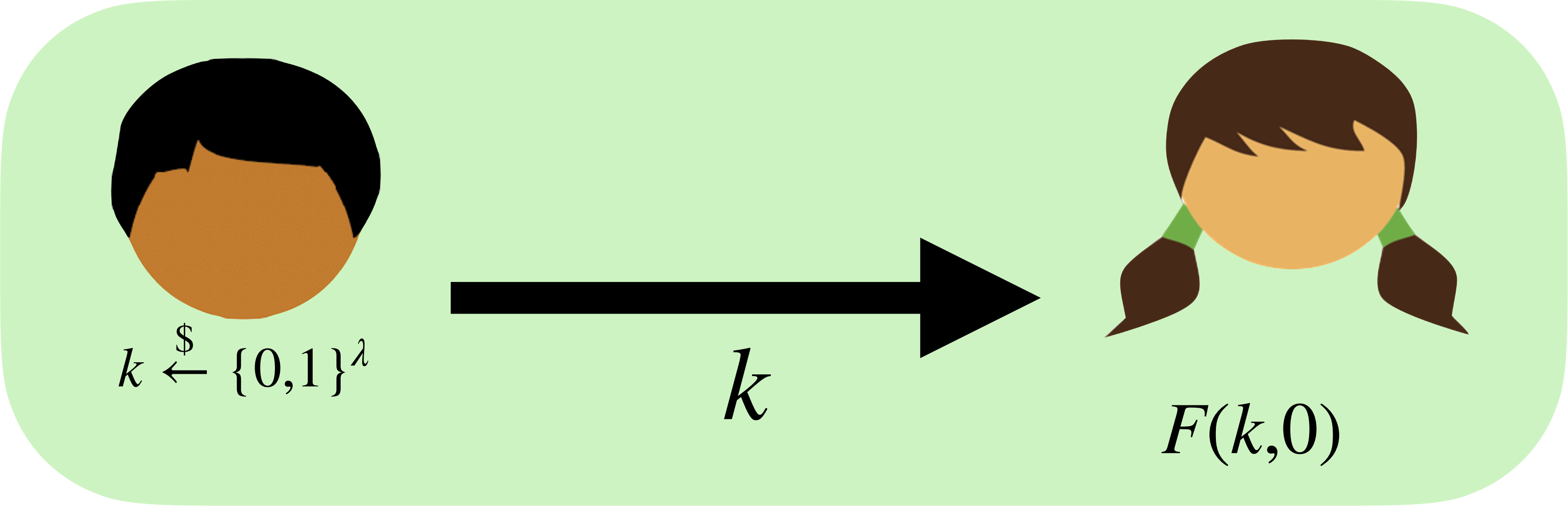
Proof: By using the existence of $\mathcal{S}_1$ to construct a distinguisher for the PRF

$$\{k, (k, F(k,0))\}$$
$$\stackrel{c}{=}$$
$$\{\mathcal{S}_1(F(k,0)), (k, F(k,0)) \mid k \leftarrow \{0,1\}^\lambda\}$$



Given $F(k,0)$, $\mathcal{S}_1$ has to spit out k

$$\{k, (k, F(k,0))\} \overset{c}{=} \{\mathcal{S}_1(F(k,0)), (k, F(k,0)) \mid k \leftarrow \{0,1\}^\lambda\}$$



$\mathscr{D}(\mathrm{PRF})$:

$m \leftarrow$ PRF.`lookup(0)`

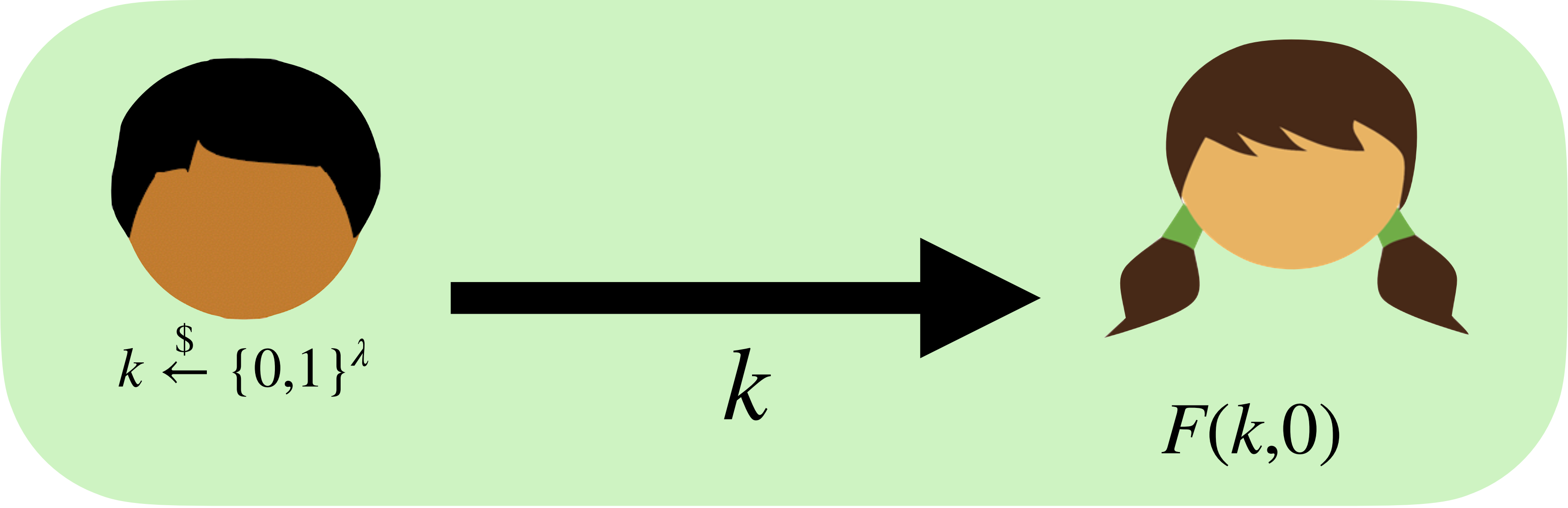
$k \leftarrow S_1(m)$

`return`

PRF.`lookup(1)` $\stackrel{?}{=} F(k,1)$

Given $F(k,0)$, $\mathcal{S}_1$ has to spit out k

$$\{k, (k, F(k,0))\}$$
$$\stackrel{c}{=}$$
$$\{\mathcal{S}_1(F(k,0)), (k, F(k,0)) \mid k \leftarrow \{0,1\}^\lambda\}$$



$\mathscr{D}$(PRF):

  $m \leftarrow$ PRF.`lookup(0)`

  $k \leftarrow S_1(m)$

  `return`

    PRF.`lookup(1)` $\stackrel{?}{=} F(k,1)$

CONTRADICTION

`Ideal:`

  $T \leftarrow$ EmptyMap

  `lookup(`m`):`

    `if` $m \notin T$:

      $T[m] \xleftarrow{\$} \{0,1\}^{\text{out}}$

    `return` $T[m]$

`Real:`

  $k \xleftarrow{\$} \{0,1\}^\lambda$

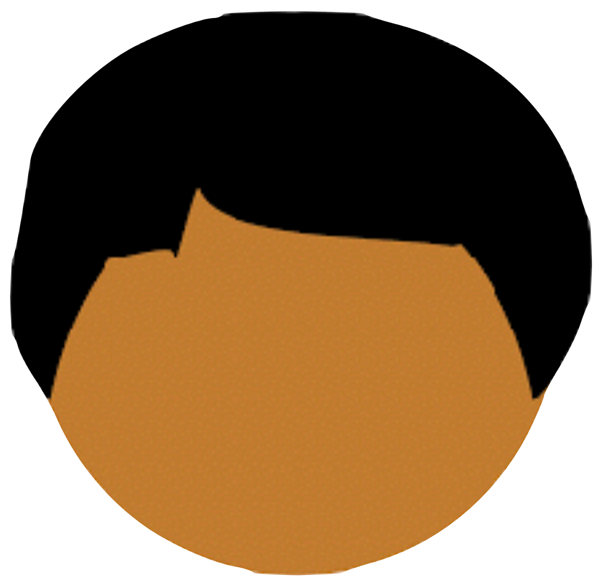
  `lookup(`m`):`

    `return` $F(k,m)$

$\stackrel{c}{\equiv}$

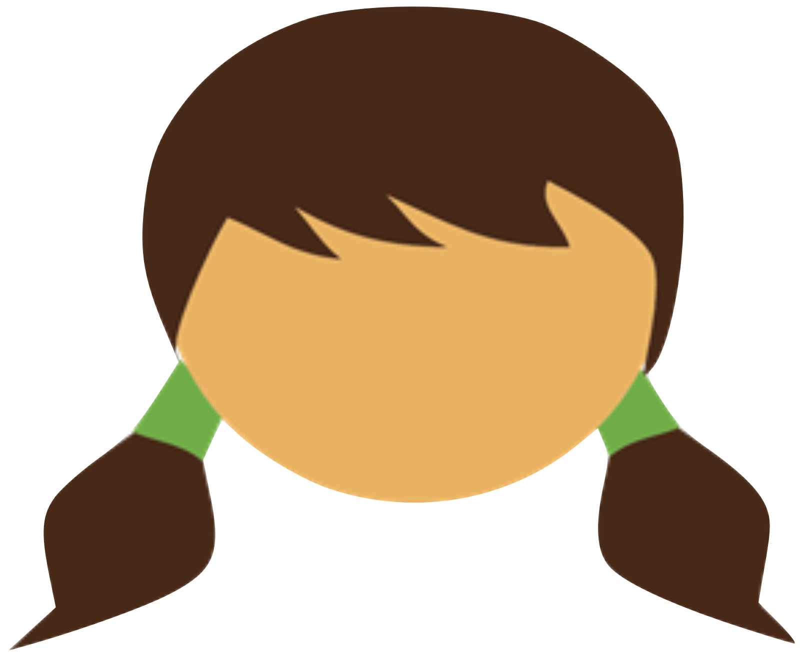
Input: P_0, P_1 input nothing

Output: P_0 outputs an encryption key k, P_1 outputs $F(k,0)$



$k \xleftarrow{\$} \{0,1\}^\lambda$

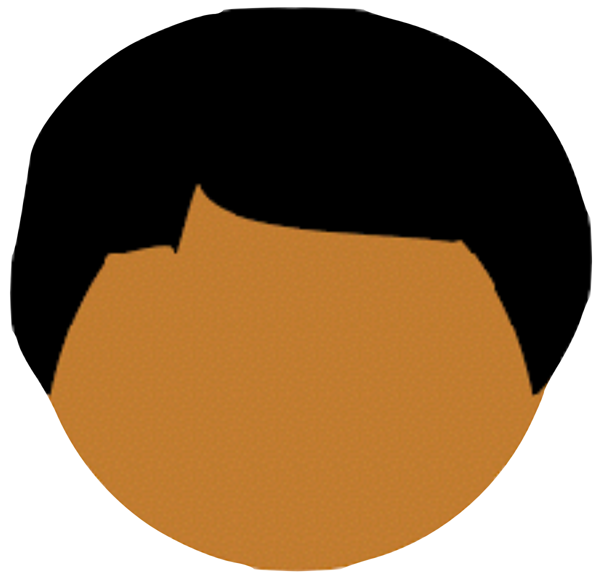
$\xrightarrow{\quad k \quad}$



$F(k,0)$

Input: P_0, P_1 input nothing

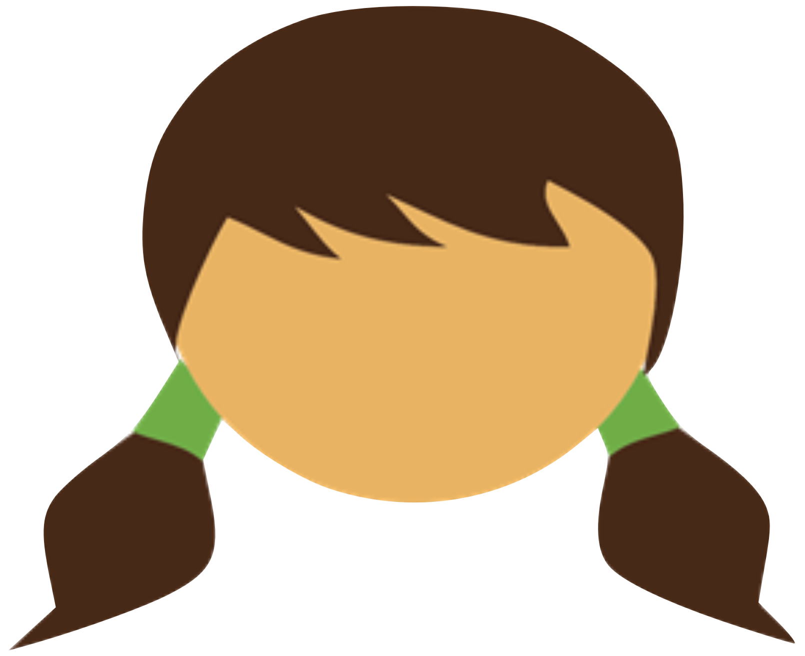
Output: P_0 outputs an encryption key k, P_1 outputs $F(k,0)$



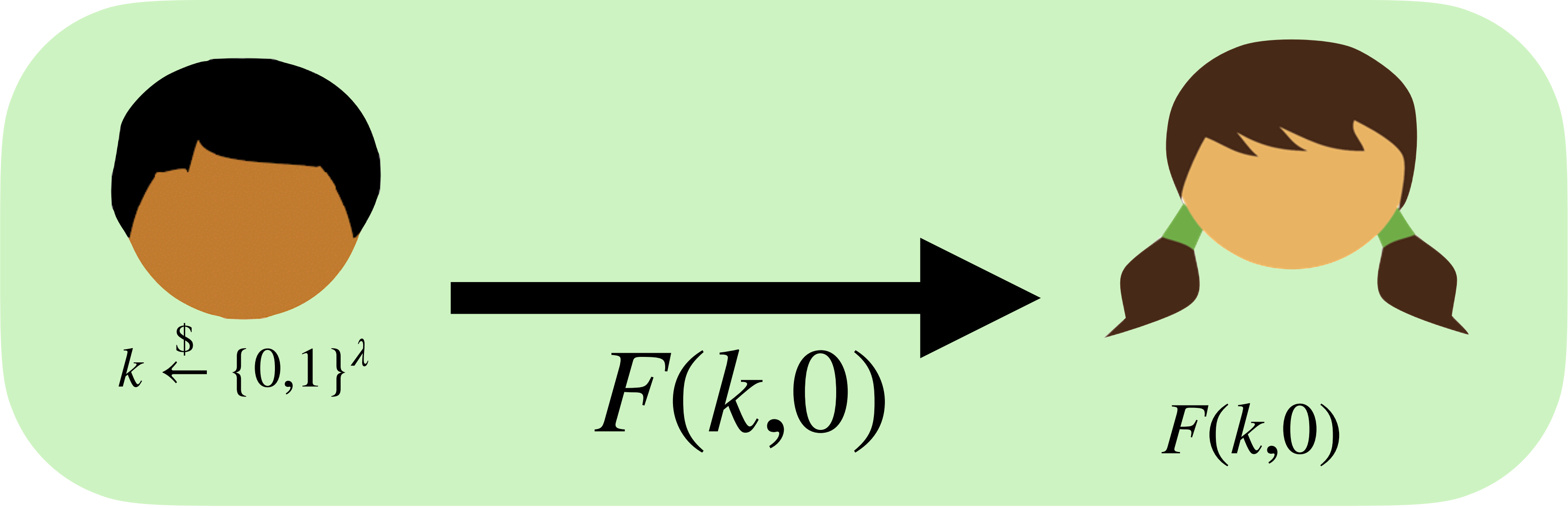
$k \xleftarrow{\$} \{0,1\}^\lambda$

$F(k,0)$

$\longrightarrow$

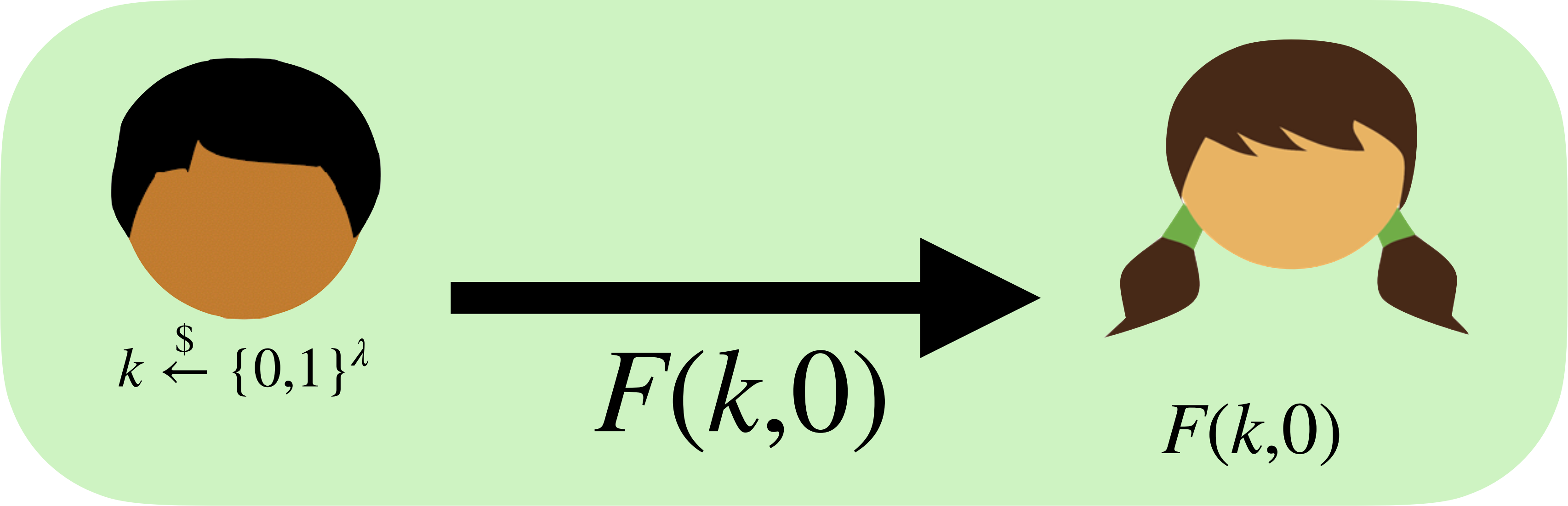


$F(k,0)$



$$\{\mathrm{View}_i^\Pi(x_0, x_1), \mathrm{Output}^\Pi(x_0, x_1)\} \overset{c}{\equiv} \{\mathcal{S}_i(x_i, y_i), (y_0, y_1) \mid (y_0, y_1) \leftarrow f(x_0, x_1)\}$$

$$\{F(k,0), (k, F(k,0))\} \overset{c}{\equiv} \{\mathcal{S}_1(F(k,0)), (k, F(k,0)) \mid k \leftarrow \{0,1\}^\lambda\}$$



$$\{\mathrm{View}_i^{\Pi}(x_0, x_1), \mathrm{Output}^{\Pi}(x_0, x_1)\}$$
$$\overset{c}{=}$$
$$\{\mathcal{S}_i(x_i, y_i), (y_0, y_1) \mid (y_0, y_1) \leftarrow f(x_0, x_1)\}$$

$$\{F(k,0), (k, F(k,0))\}$$
$$\overset{c}{=}$$
$$\{\mathcal{S}_1(F(k,0)), (k, F(k,0)) \mid k \leftarrow \{0,1\}^\lambda\}$$

$\mathcal{S}_1(F(k,0))$:
    return $F(k,0)$

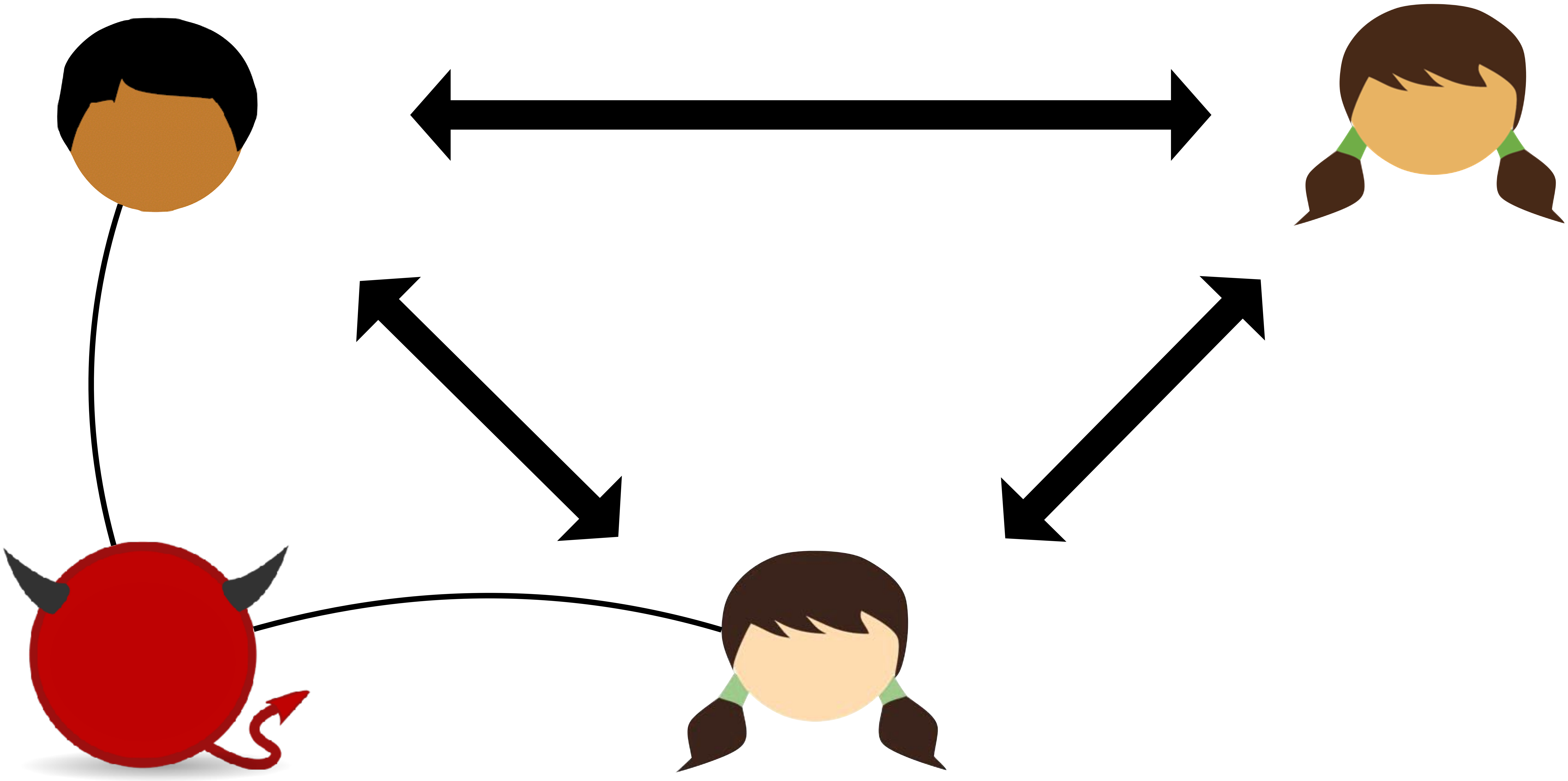
Two-Party Semi-Honest Security

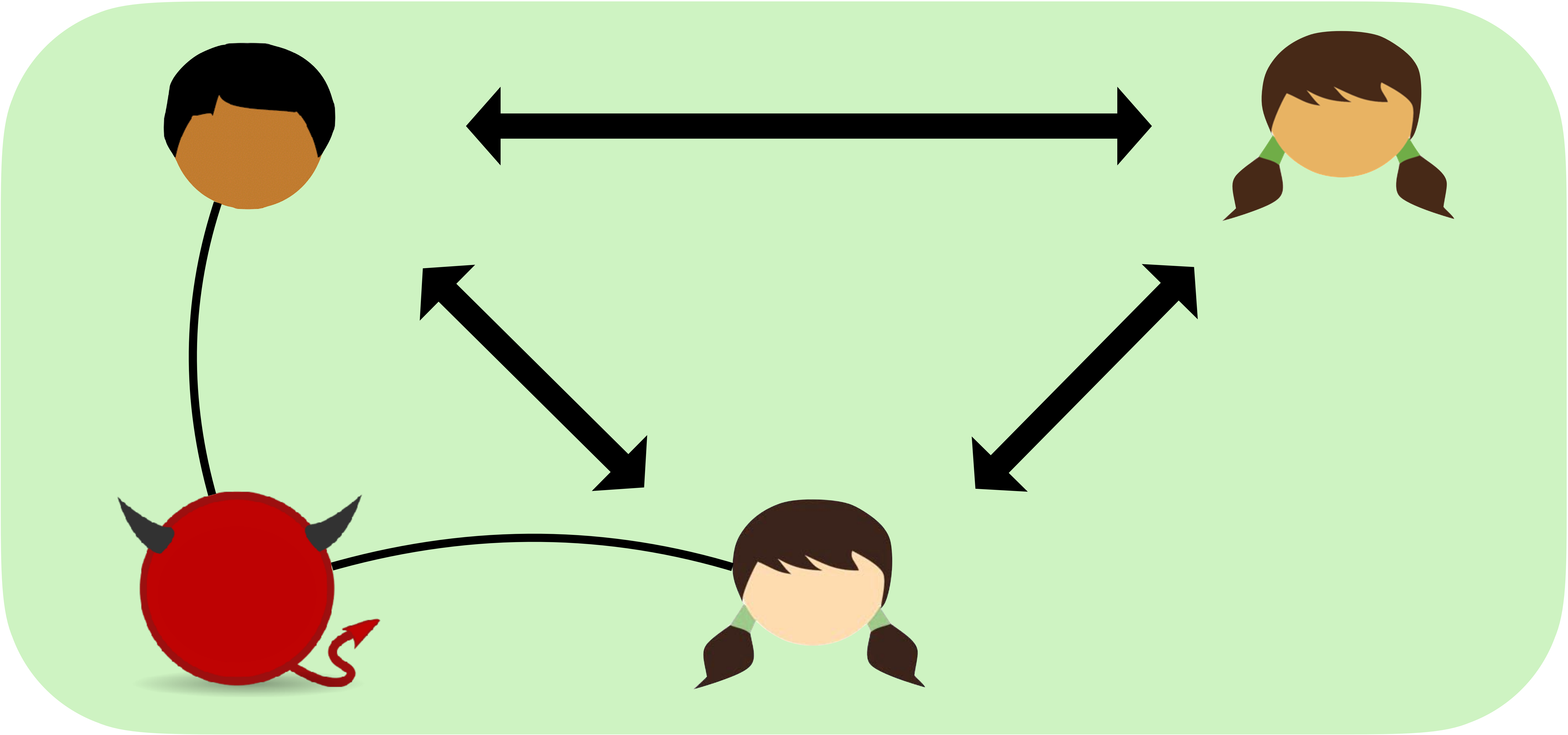
Let f *be a functionality. We say that a protocol* Π *securely computes* f *in the presence of a semi-honest adversary if for each party* $i \in \{0,1\}$ *there exists a polynomial time* simulator $\mathcal{S}_i$ *such that for all inputs* x_0, x_1*:*

$$\{\text{View}_i^{\Pi}(x_0, x_1), \text{Output}^{\Pi}(x_0, x_1)\}$$

$$\overset{c}{=}$$

$$\{\mathcal{S}_i(x_i, y_i), (y_0, y_1) \mid (y_0, y_1) \leftarrow f(x_0, x_1)\}$$





We consider a single global adversary who corrupts a subset of the parties

Two-Party Semi-Honest Security

Let f be a functionality. We say that a protocol Π securely computes f in the presence of a semi-honest adversary if for each party $i \in \{0,1\}$ there exists a polynomial time simulator $\mathcal{S}_i$ *such that for all inputs x_0, x_1:*

$$\{\text{View}_i^{\Pi}(x_0, x_1), \text{Output}^{\Pi}(x_0, x_1)\}$$

$$\approx$$

$$\{\mathcal{S}_i(x_i, y_i), (y_0, y_1) \mid (y_0, y_1) \leftarrow f(x_0, x_1)\}$$

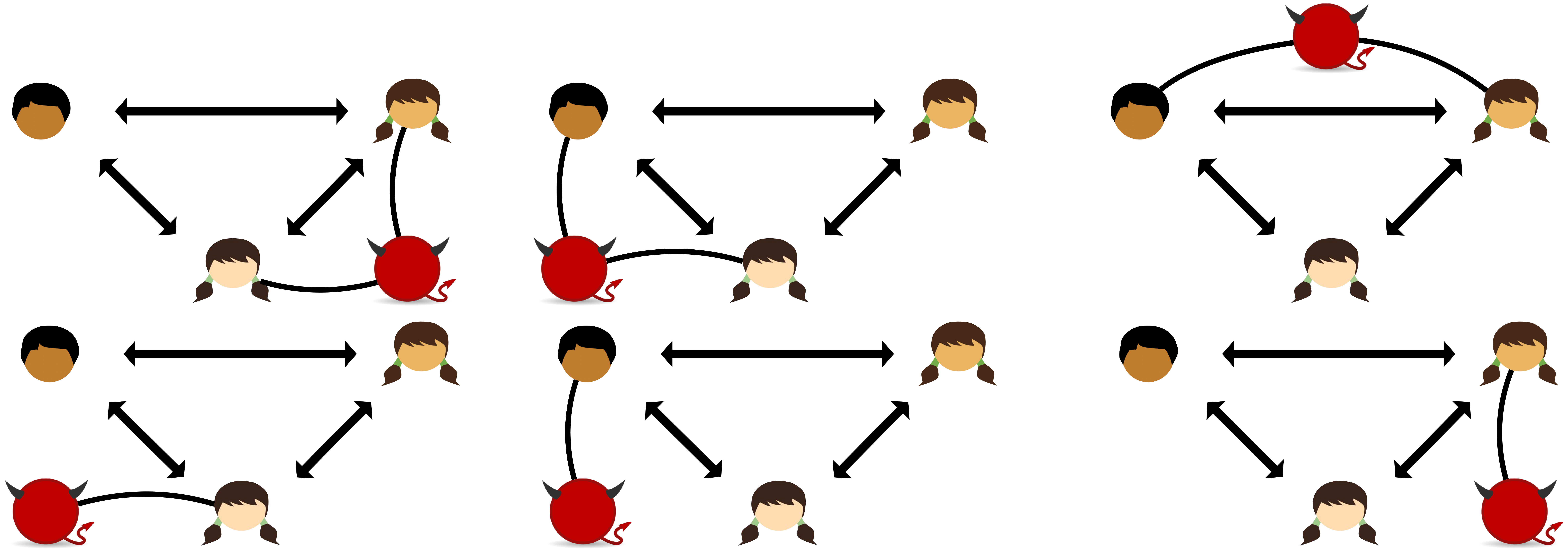
Semi-Honest Security

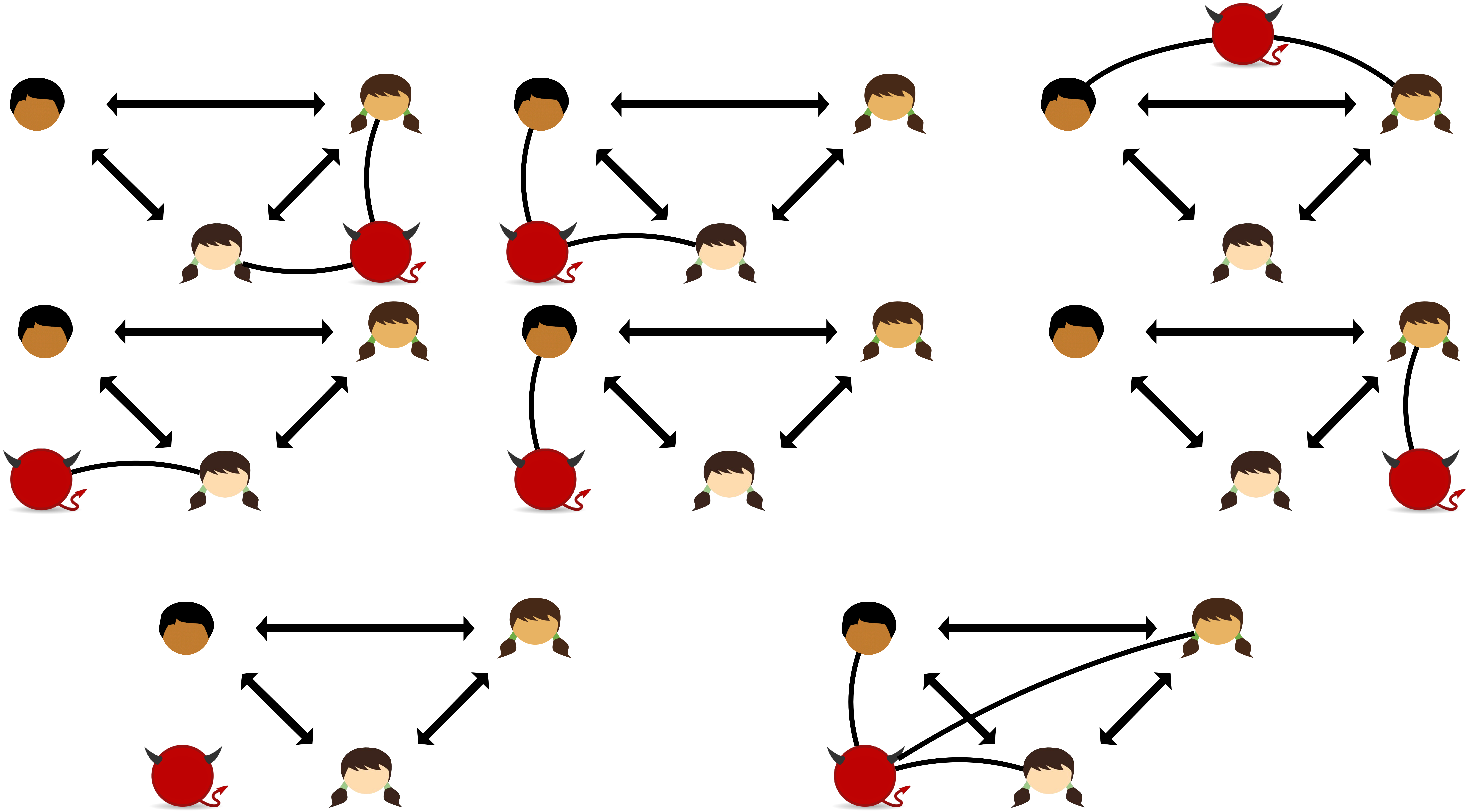
Let $P_0, \ldots, P_{n-1}$ be n parties. Let f be a functionality. We say that a protocol Π securely computes f in the presence of a semi-honest adversary if for each subset $c \subseteq \{0,...,n-1\}$ of corrupted parties there exists a polynomial time simulator $\mathcal{S}_c$ *such that for all inputs $x_0, \ldots, x_{n-1}$:*

$$\left\{ \left(\bigcup_{i \in c} \mathrm{View}_i^{\Pi}(x_0, \ldots, x_{n-1}) \right), \mathrm{Output}^{\Pi}(x_0, \ldots, x_{n-1}) \right\}$$

$$\approx$$

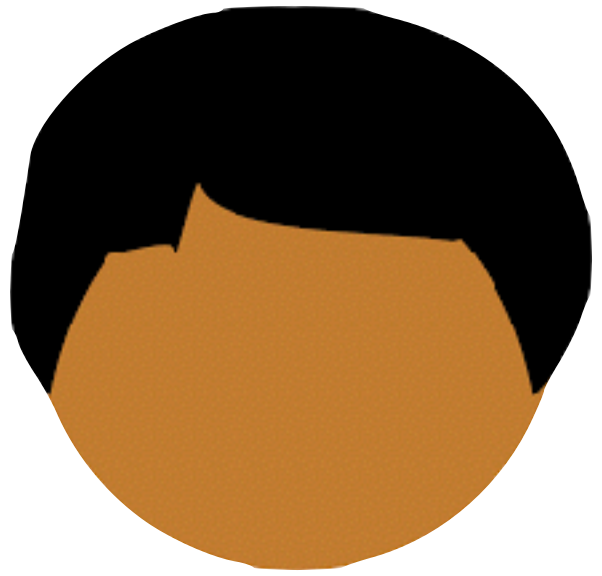
$$\left\{ \mathcal{S}_c \left(\bigcup_{i \in c} \{x_i, y_i\} \right), (y_0, \ldots y_{n-1}) \mid (y_0, \ldots y_{n-1}) \leftarrow f(x_0, \ldots, x_{n-1}) \right\}$$

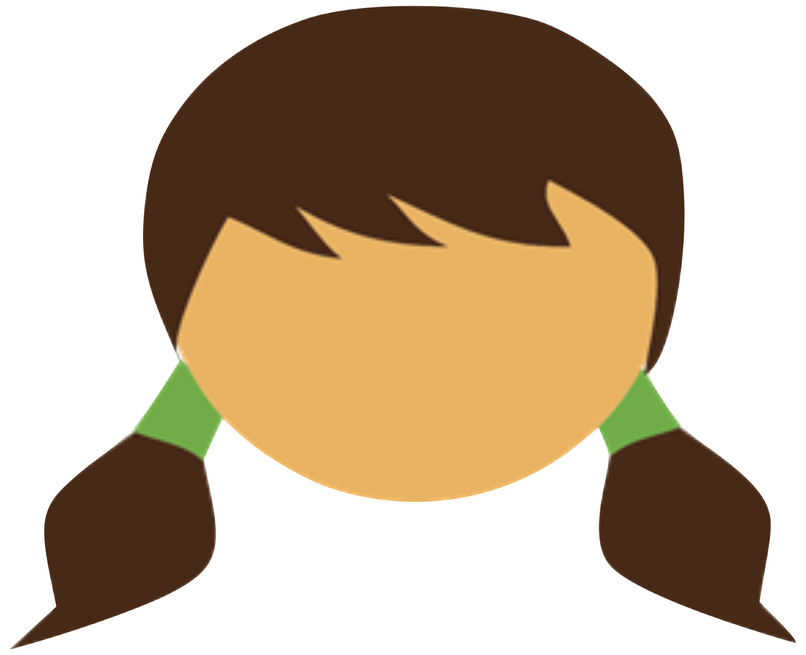




Multiparty GMW

XOR Secret Shares

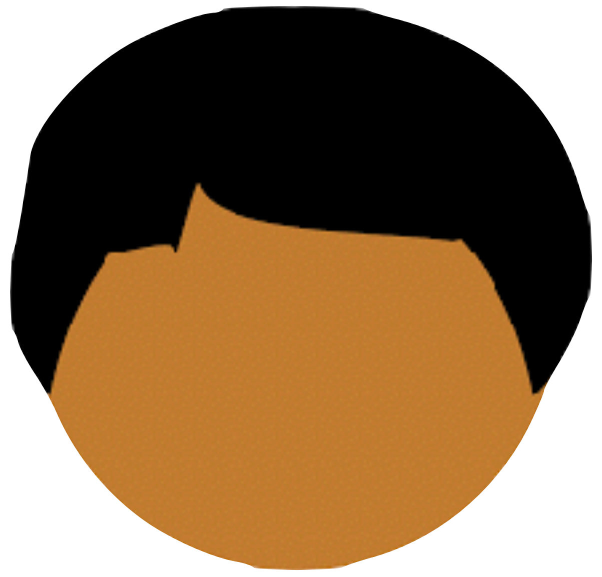


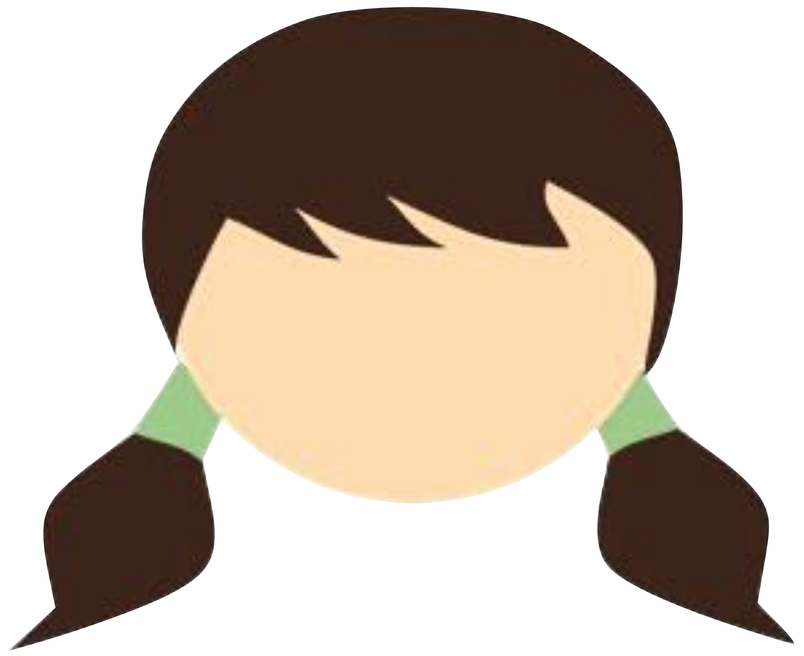


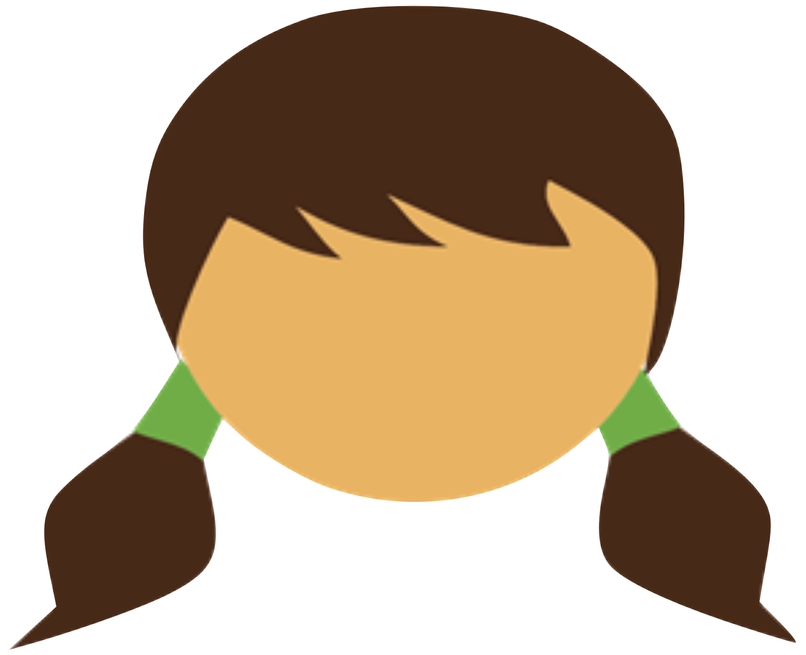
The XOR secret sharing *of a bit x is a pair of bits $\langle x_0, x_1 \rangle$ where P_0 holds x_0 and P_1 holds x_1, and where $x_0 \oplus x_1 = x$*

We sometimes denote such a pair by $[x]$

XOR Secret Shares



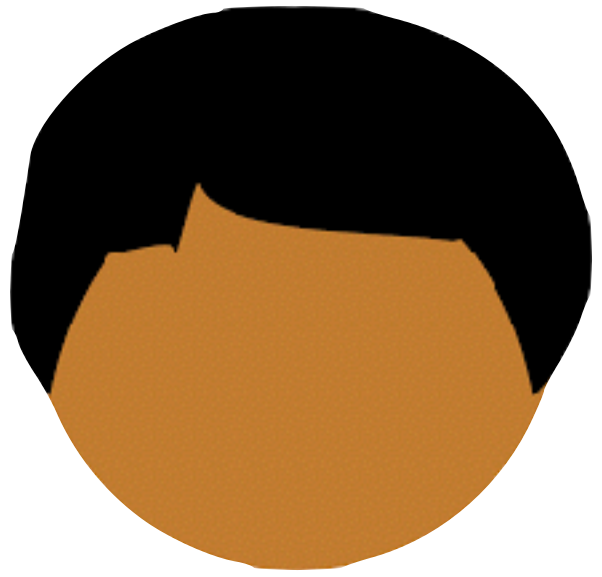




The XOR secret sharing *of a bit x is a tuple of bits $\langle x_0, \ldots, x_{n-1} \rangle$ where P_i holds x_i, and where:*

$$\left(\bigoplus_i x_i \right) = x$$

We sometimes denote such a pair by $[x]$

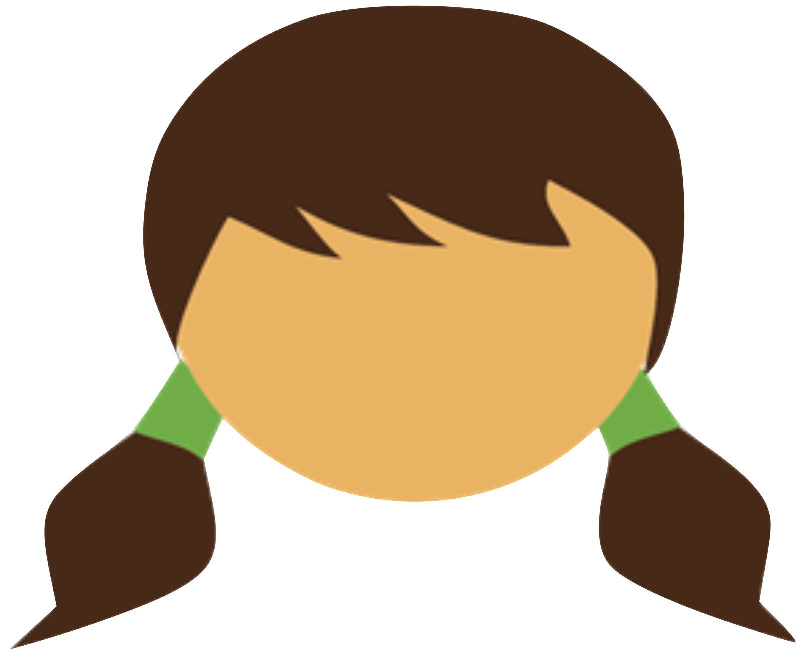


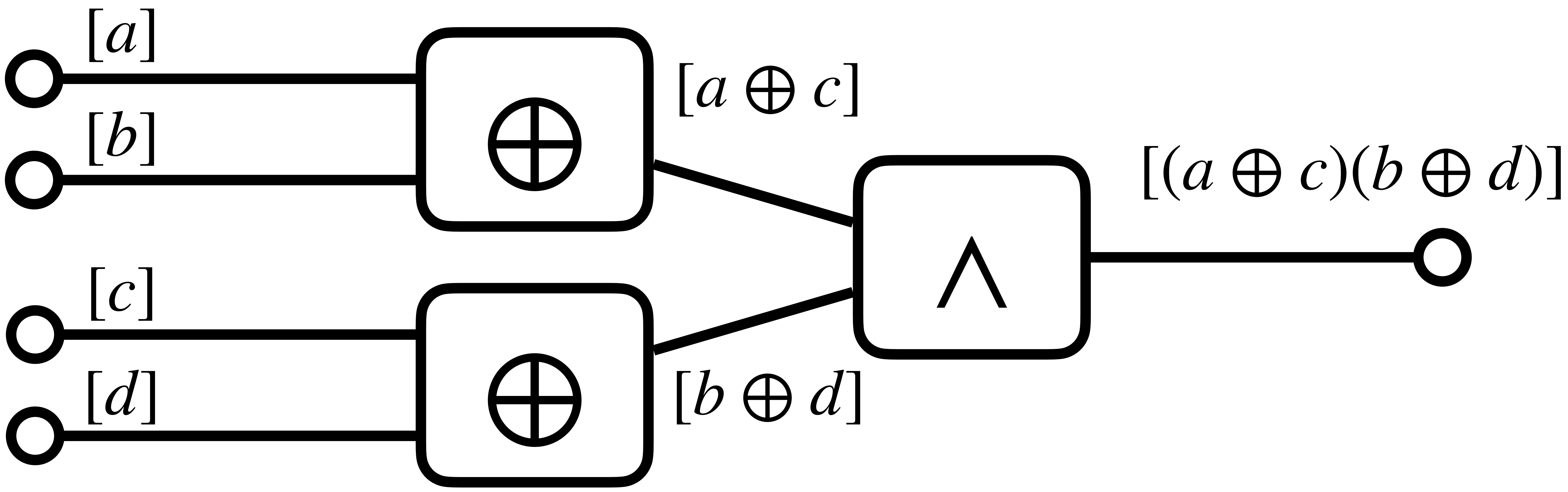
Where do input shares come from?

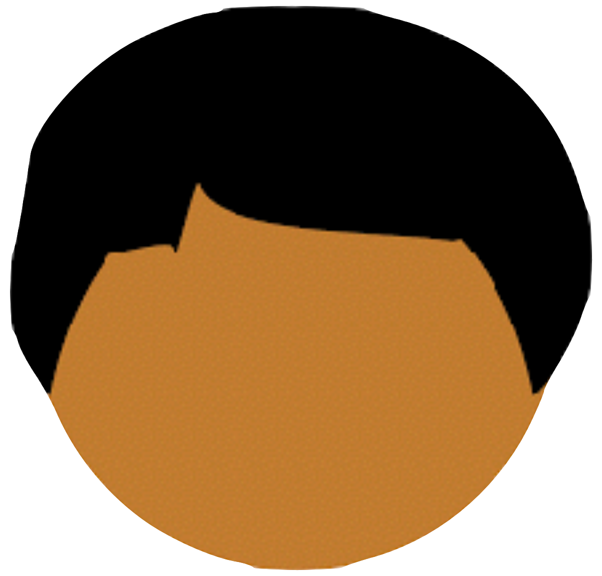
How do we XOR two shares?

How do we AND two shares?

How do we “decrypt” output shares?





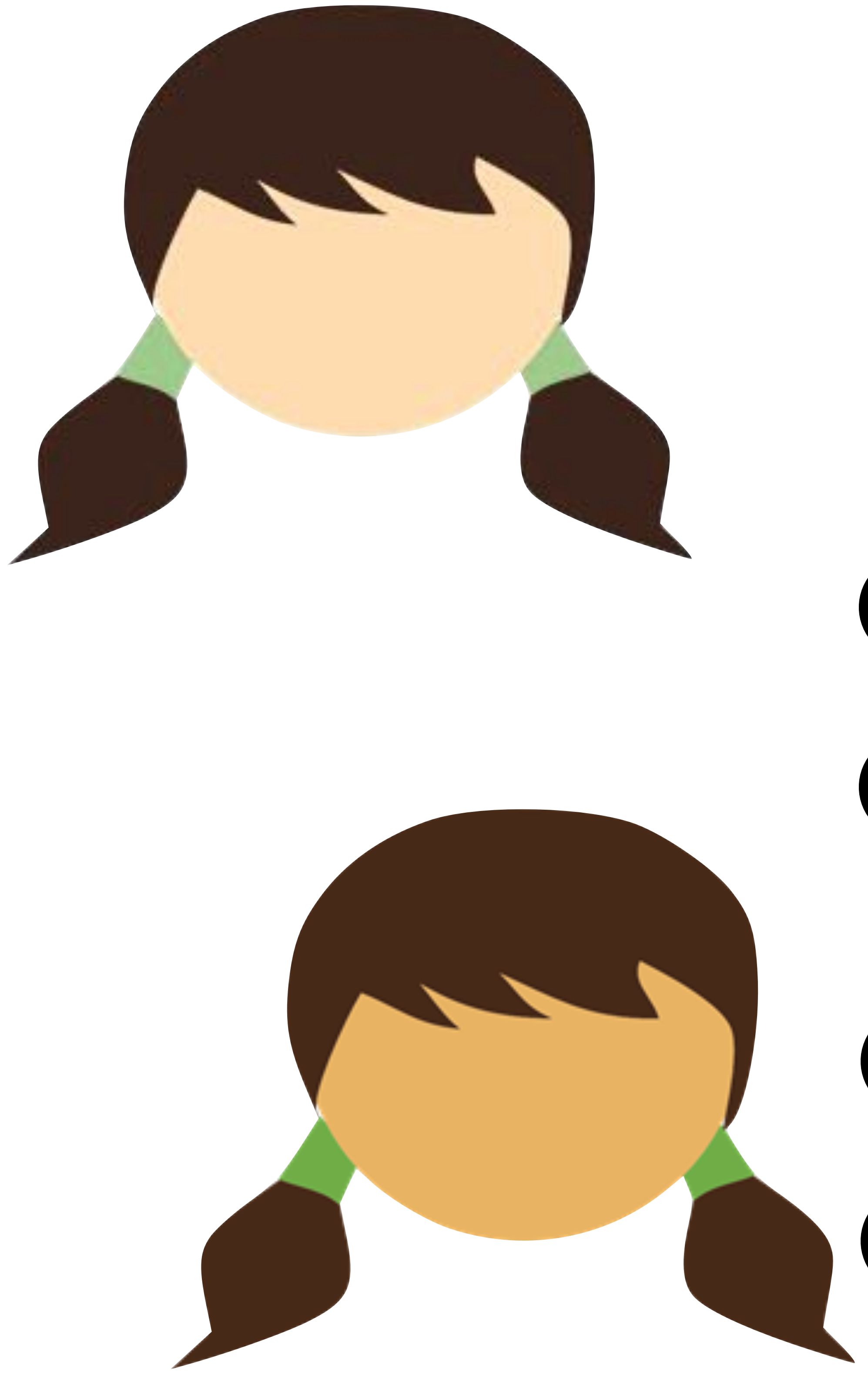


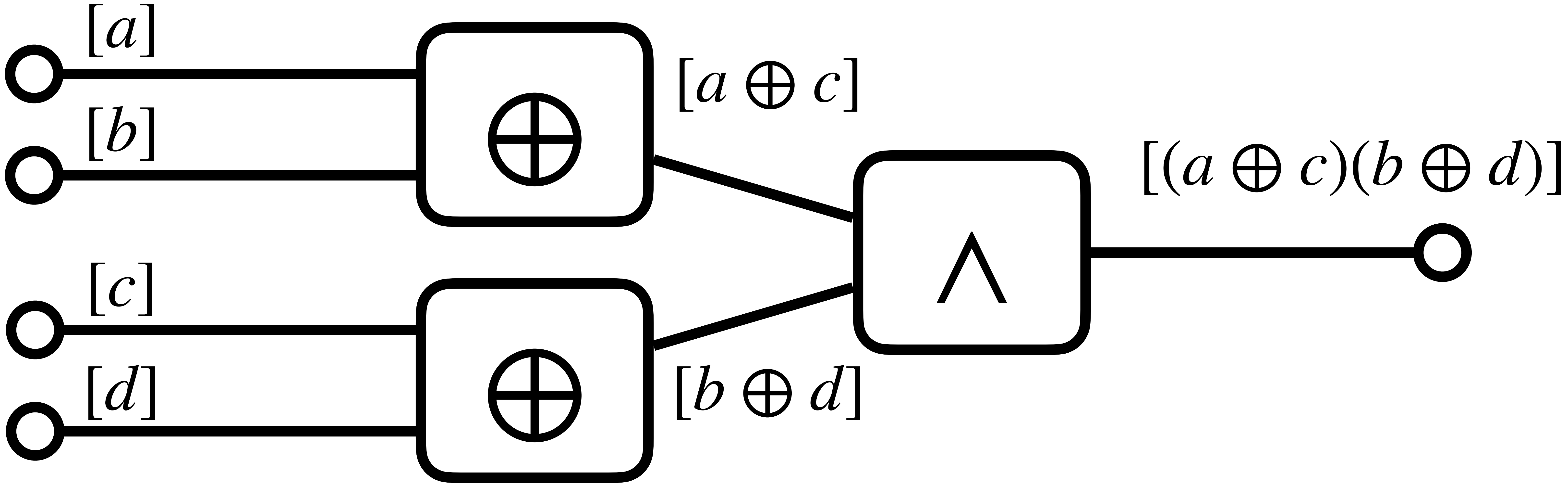
Where do input shares come from?

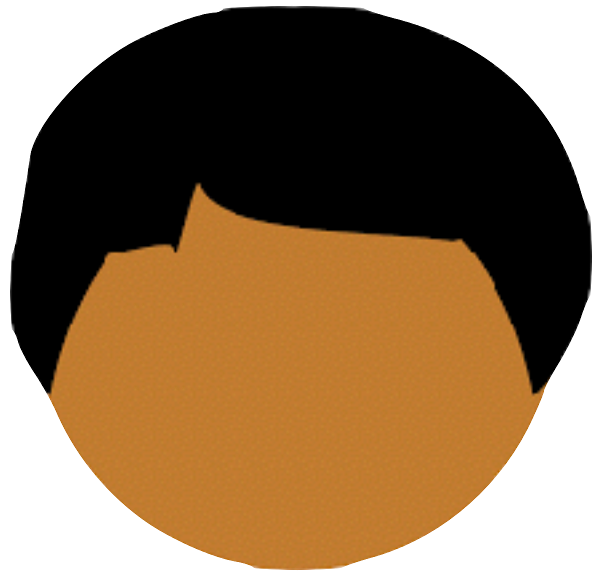
How do we XOR two shares?

How do we AND two shares?

How do we “decrypt” output shares?





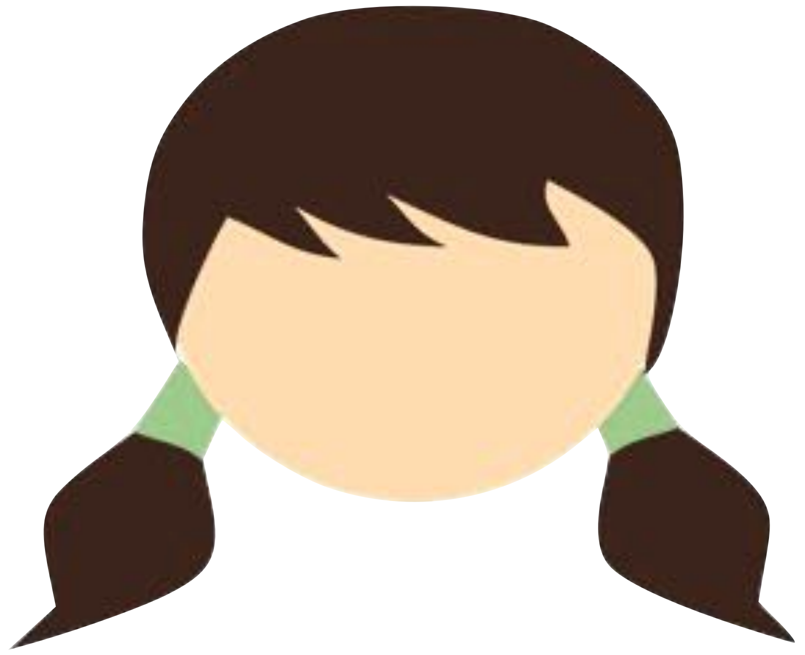


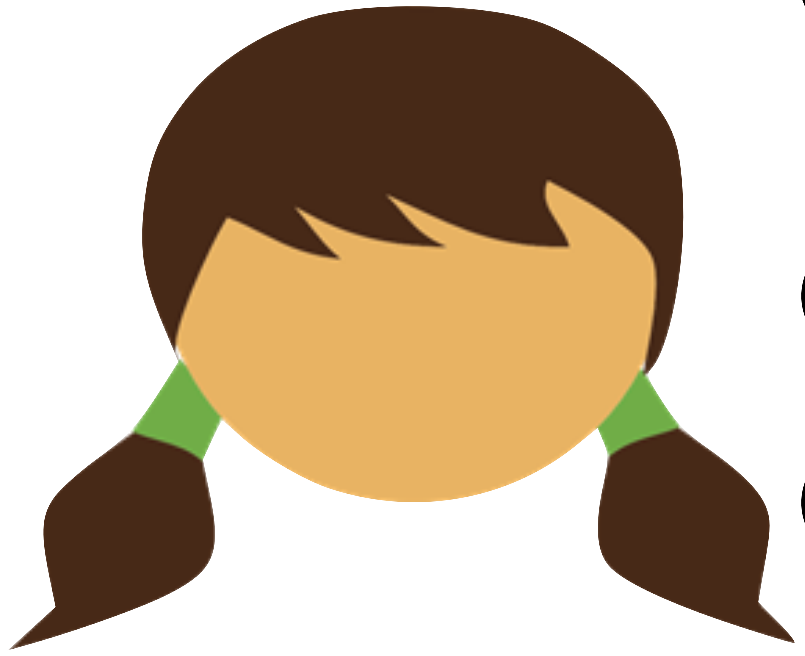
Where do input shares come from? ✓

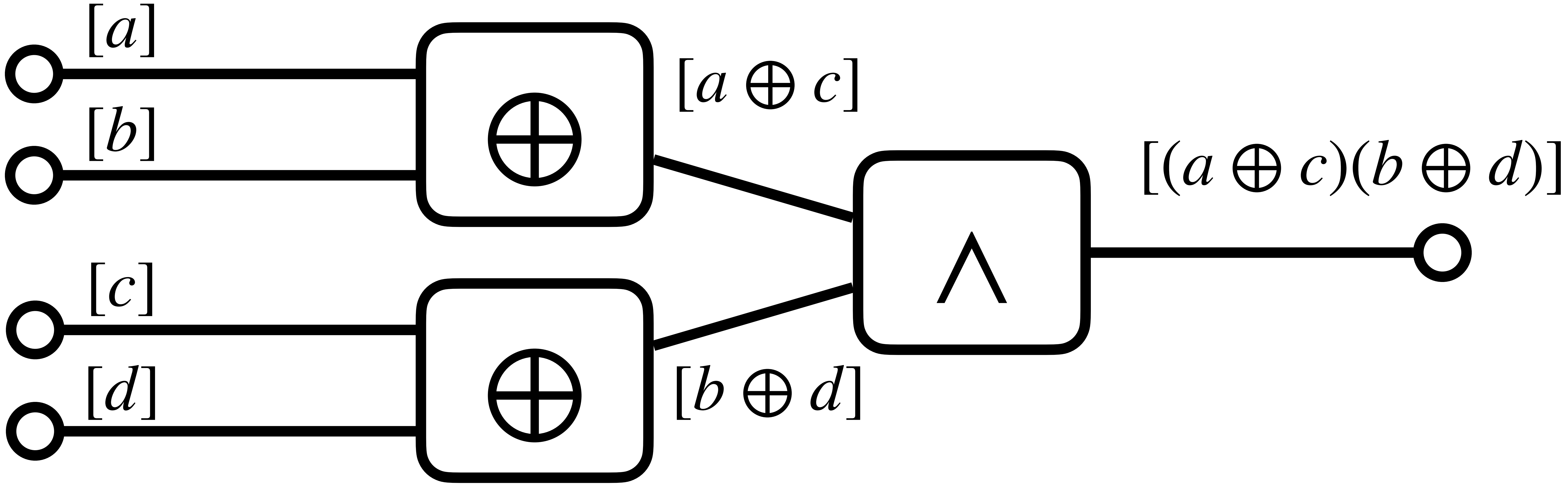
How do we XOR two shares? ✓

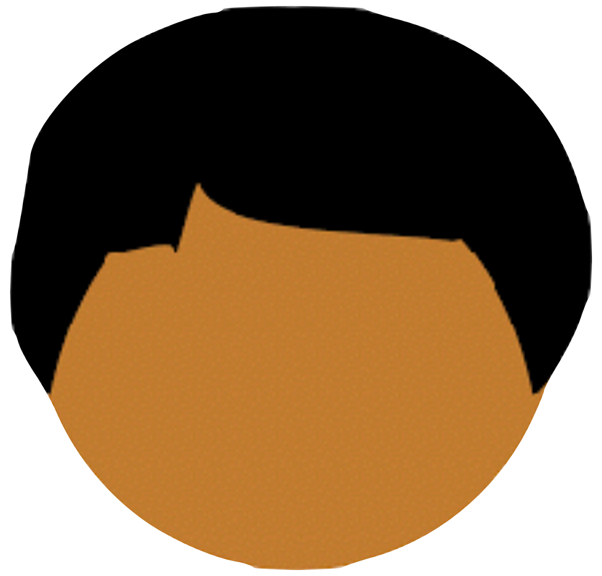
How do we AND two shares?

How do we "decrypt" output shares? ✓

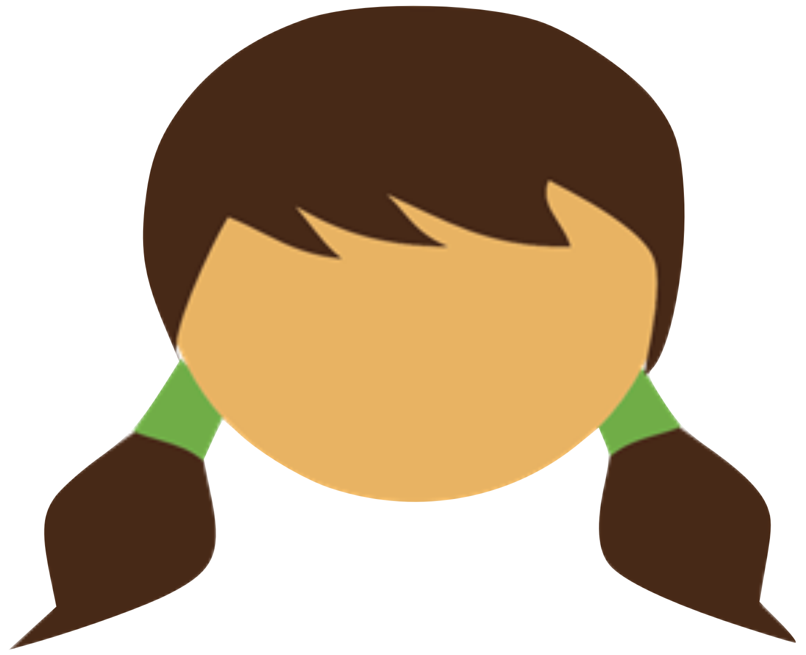








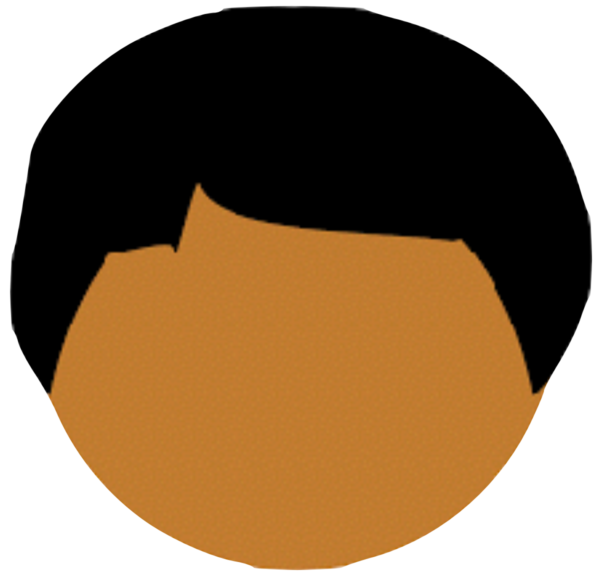
How do we AND two shares?



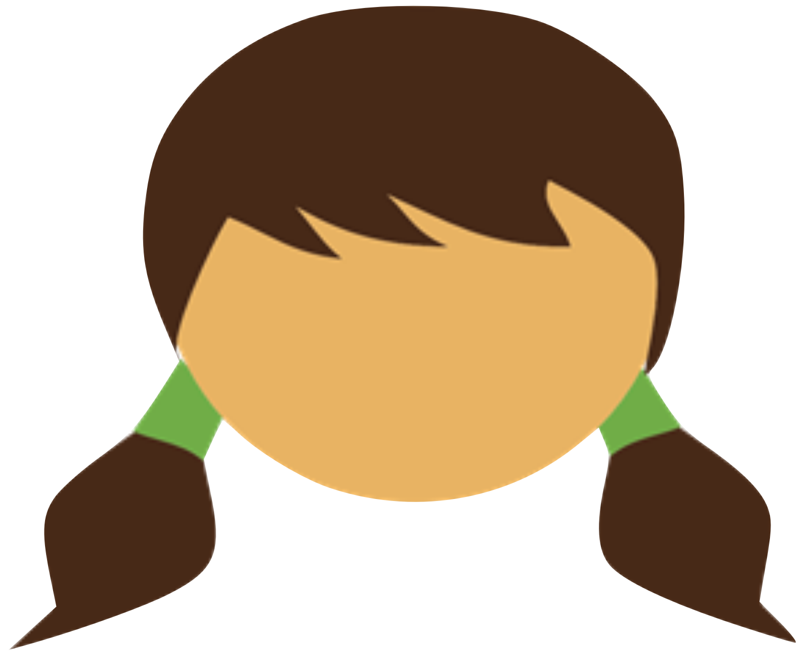
Goal: given gate input wires holding $[x]$, $[y]$, put $[x \wedge y]$ on the gate output

$$(x_0 \oplus x_1) \wedge (y_0 \oplus y_1)$$

$$= (x_0 \wedge y_0) \oplus (x_0 \wedge y_1) \oplus (x_1 \wedge y_0) \oplus (x_1 \wedge y_1)$$



How do we AND two shares?

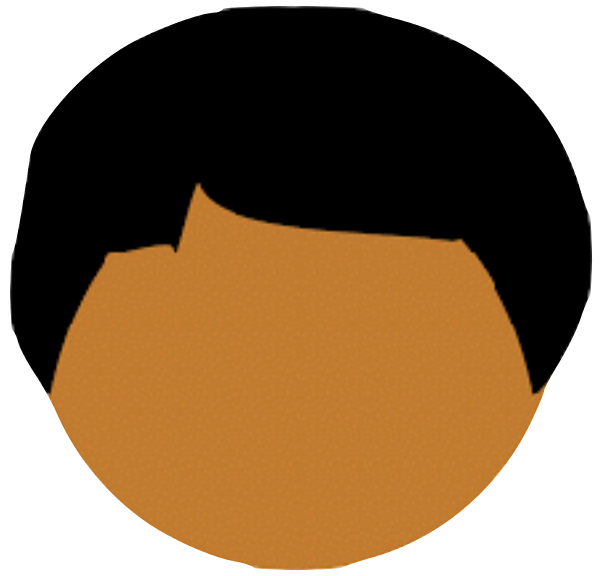


Goal: given gate input wires holding $[x]$, $[y]$, put $[x \wedge y]$ on the gate output

$$(x_0 \oplus x_1) \wedge (y_0 \oplus y_1)$$

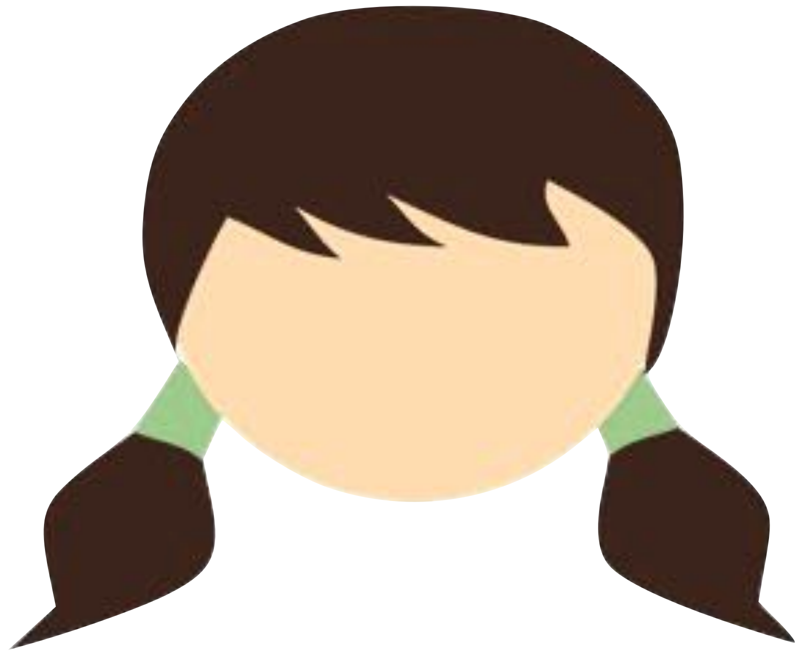
$$= (x_0 \wedge y_0) \oplus (x_0 \wedge y_1) \oplus (x_1 \wedge y_0) \oplus (x_1 \wedge y_1)$$

OT

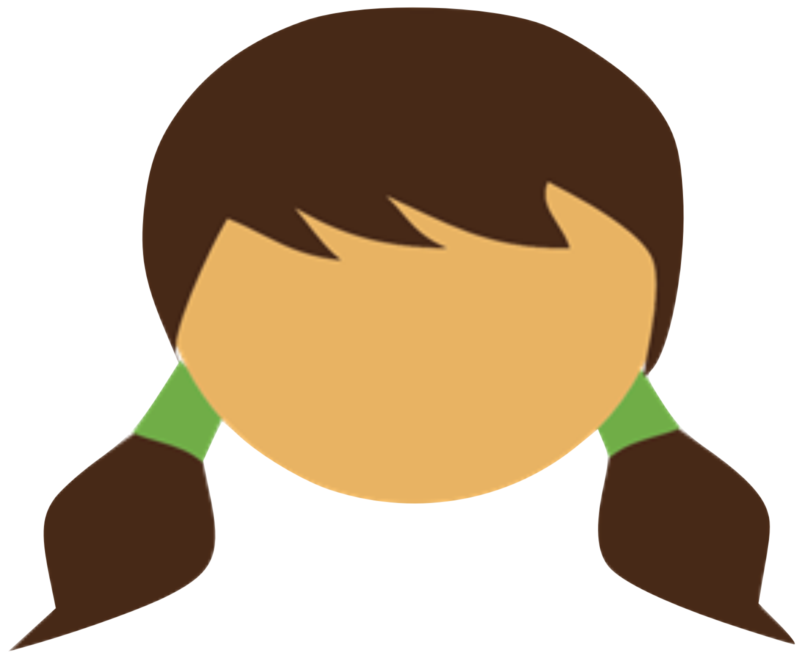


How do we AND two shares?

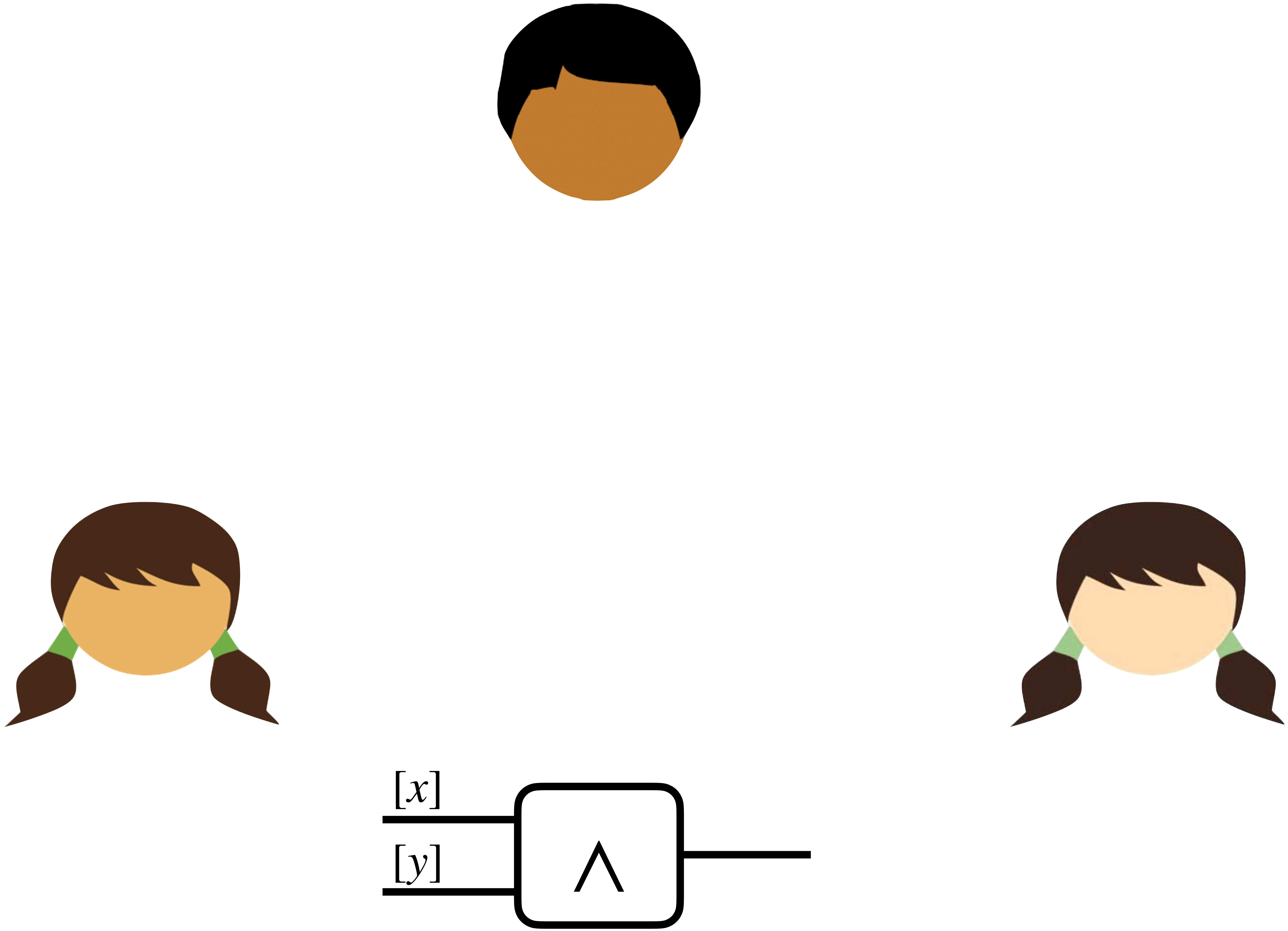
Goal: given gate input wires holding $[x]$, $[y]$, put $[x \wedge y]$ on the gate output

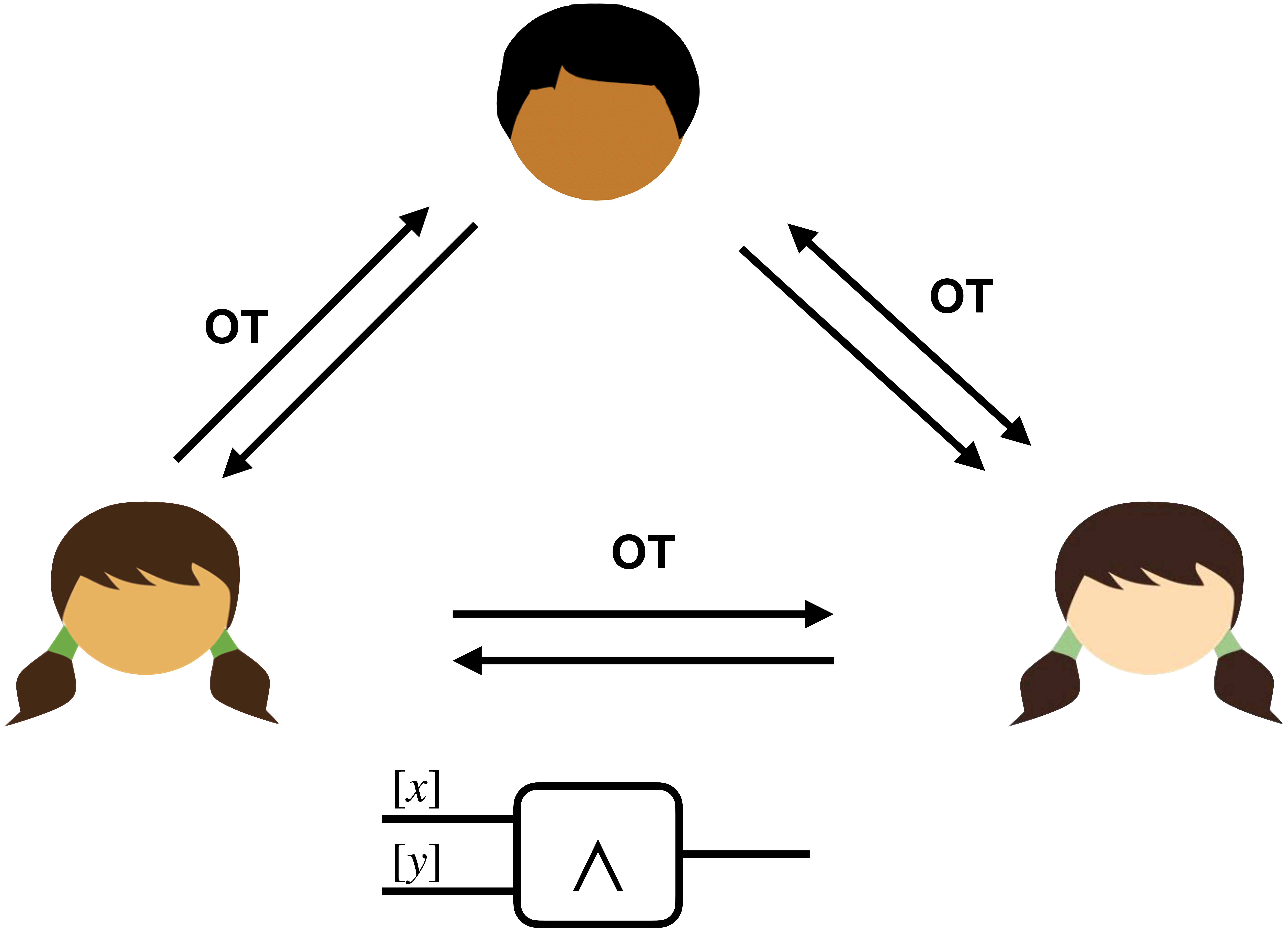


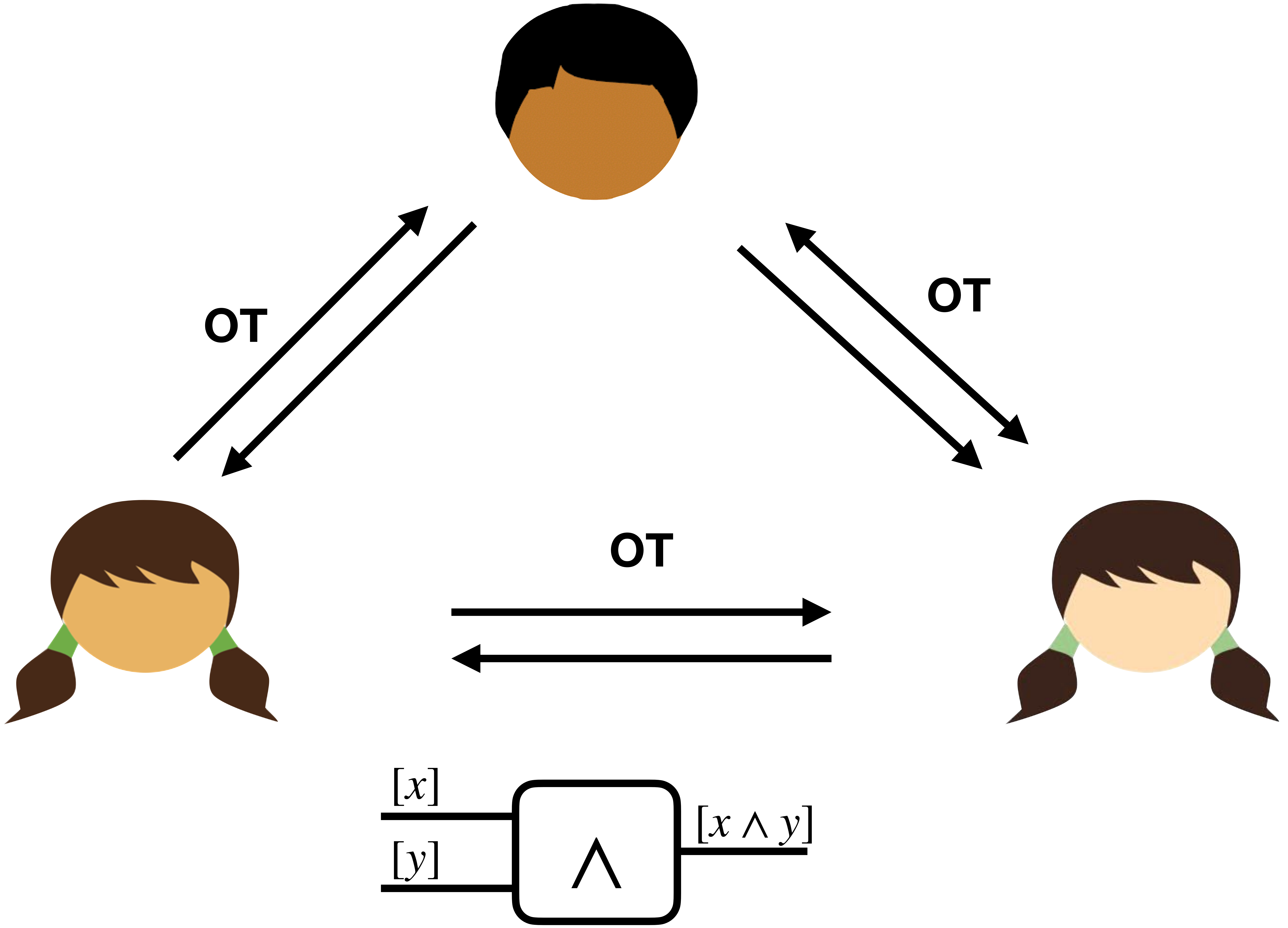
$$\left(\bigoplus_i x_i\right) \wedge \left(\bigoplus_i y_i\right)$$



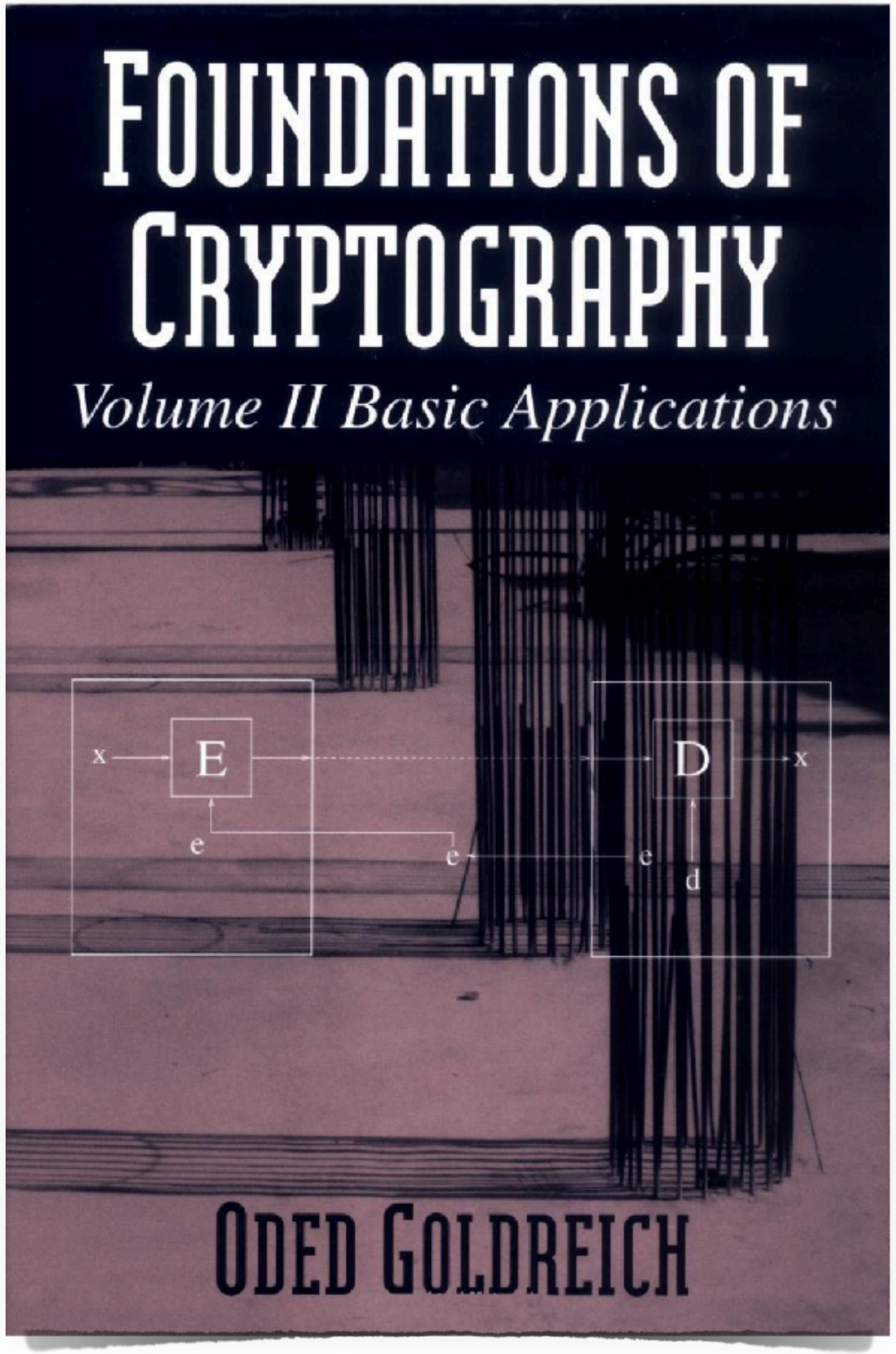
$$\bigoplus_{i,j} x_i \wedge y_j$$


[x]
[y]
∧


OT
OT
OT
[x]
[y]
∧


OT
OT
OT
[x]
[y]
∧
[x ∧ y]

GMW Security



Theorem 7.3.3 (Composition Theorem for the semi-honest model): *Suppose that g is privately reducible to f and that there exists a protocol for privately computing f. Then there exists a protocol for privately computing g.*

Composition

Suppose we have a protocol ρ *that securely computes a functionality* g

Composition

Suppose we have a protocol ρ *that securely computes a functionality* g

Suppose we write a new a "hybrid" protocol π *that uses* g *as a black box*

Composition

Suppose we have a protocol ρ *that securely computes a functionality* g

Suppose we write a new a "hybrid" protocol π *that uses* g *as a black box*

Now we prove π *securely computes* f *when using* g *as a black box*

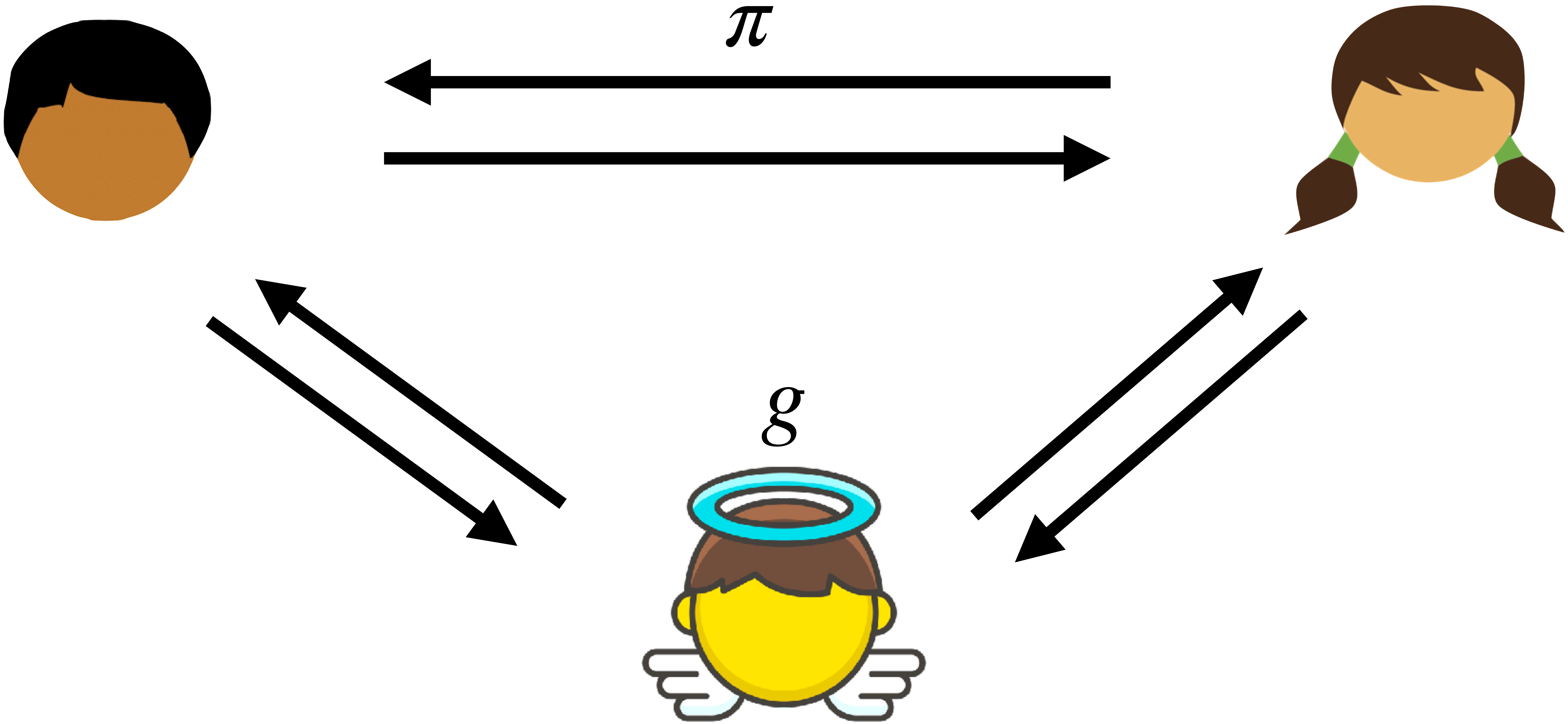
Composition

Suppose we have a protocol ρ that securely computes a functionality g

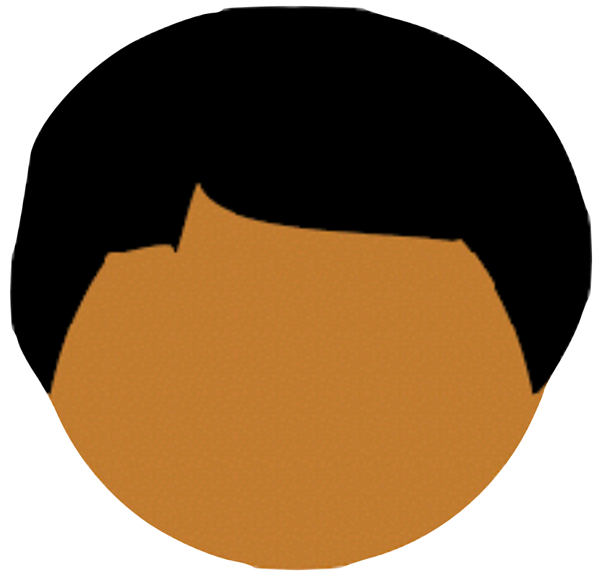
Suppose we write a new a "hybrid" protocol π that uses g as a black box

Now we prove π securely computes f when using g as a black box

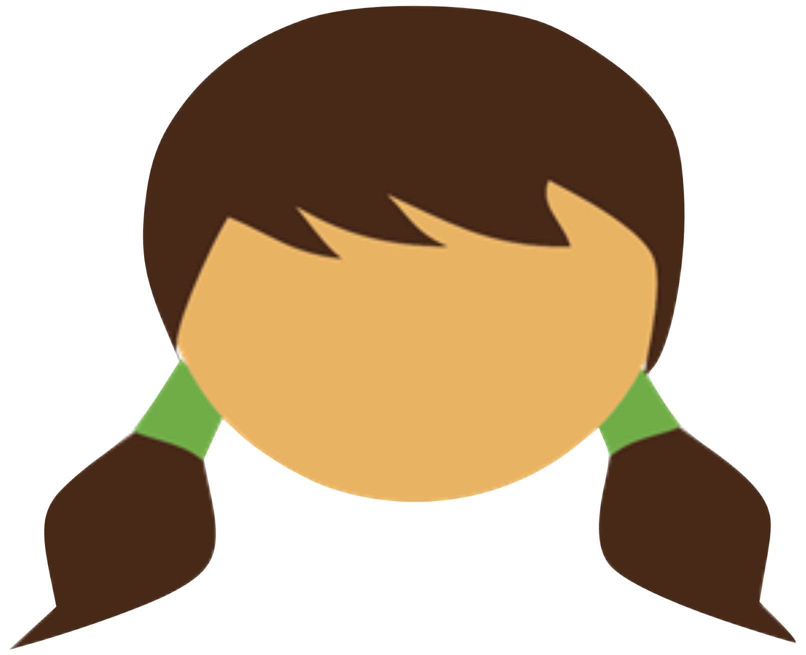
If we then substitute calls to g by ρ, then the resulting protocol securely implements f


π
g


ρ
≈
g
π
g
≈
π with ρ

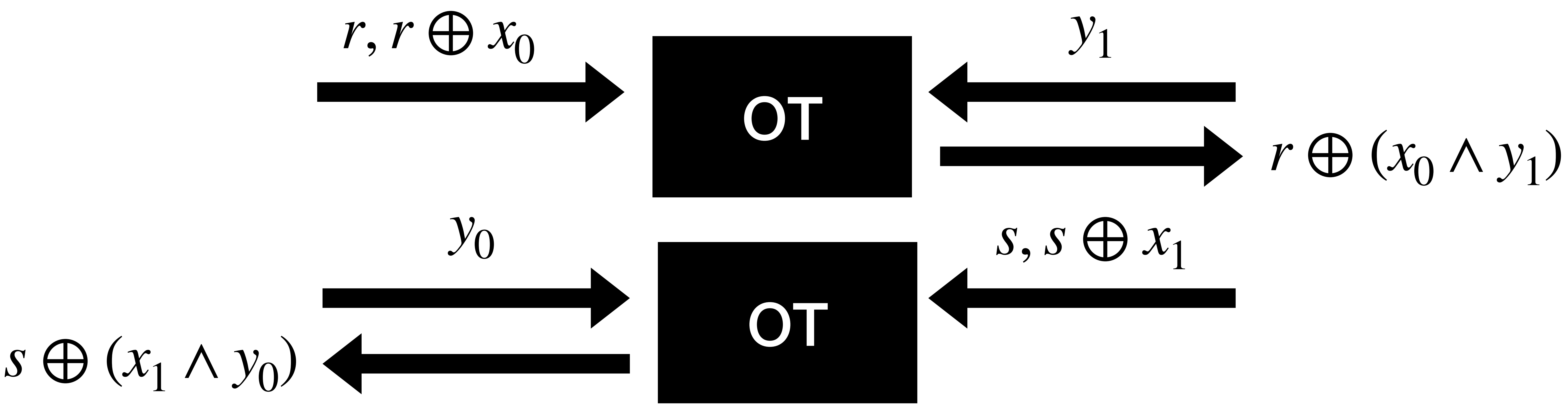


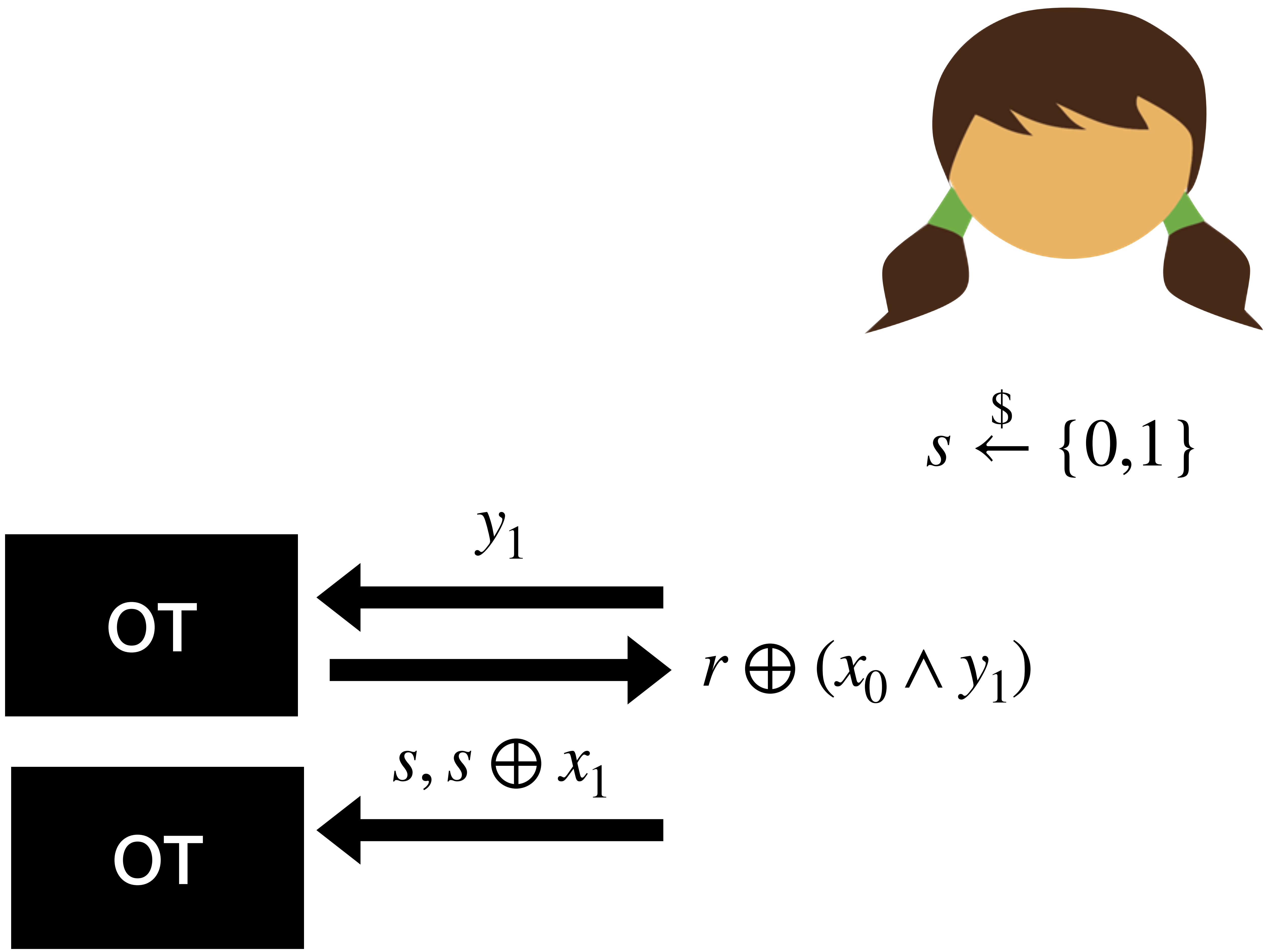
Goal: given gate input wires holding $[x], [y]$, put $[x \wedge y]$ on the gate output

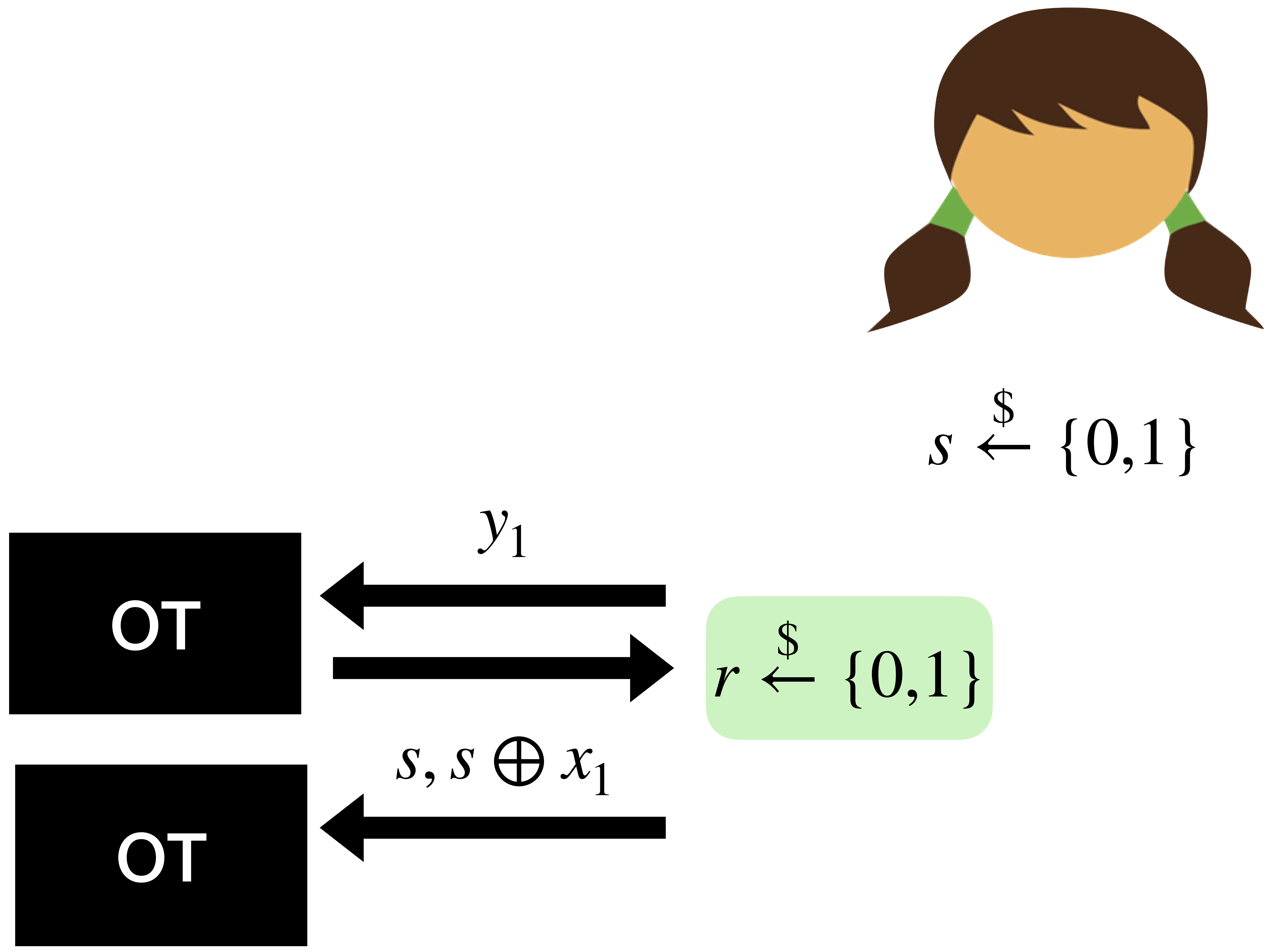


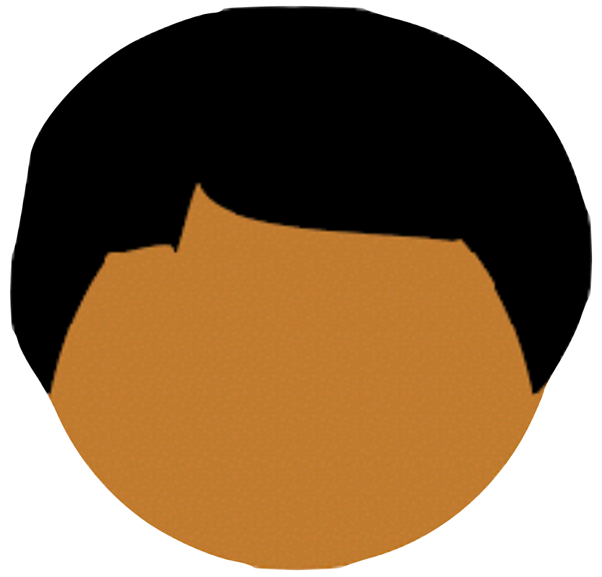
$r \xleftarrow{\$} \{0,1\}$

$s \xleftarrow{\$} \{0,1\}$




$s \xleftarrow{\$} \{0,1\}$
y_1
OT
$r \oplus (x_0 \wedge y_1)$
$s, s \oplus x_1$
OT


$s \xleftarrow{\$} \{0,1\}$
y_1
OT
$r \xleftarrow{\$} \{0,1\}$
$s, s \oplus x_1$
OT

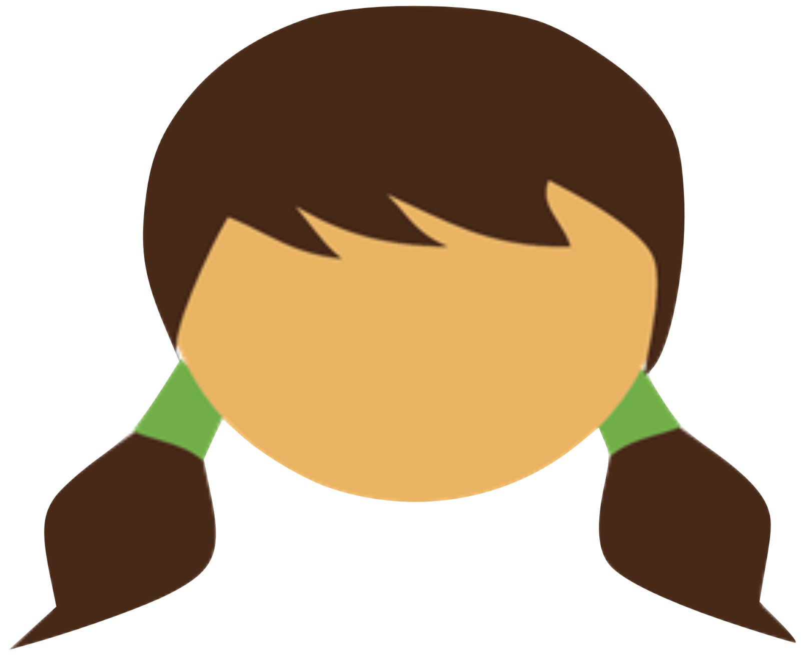


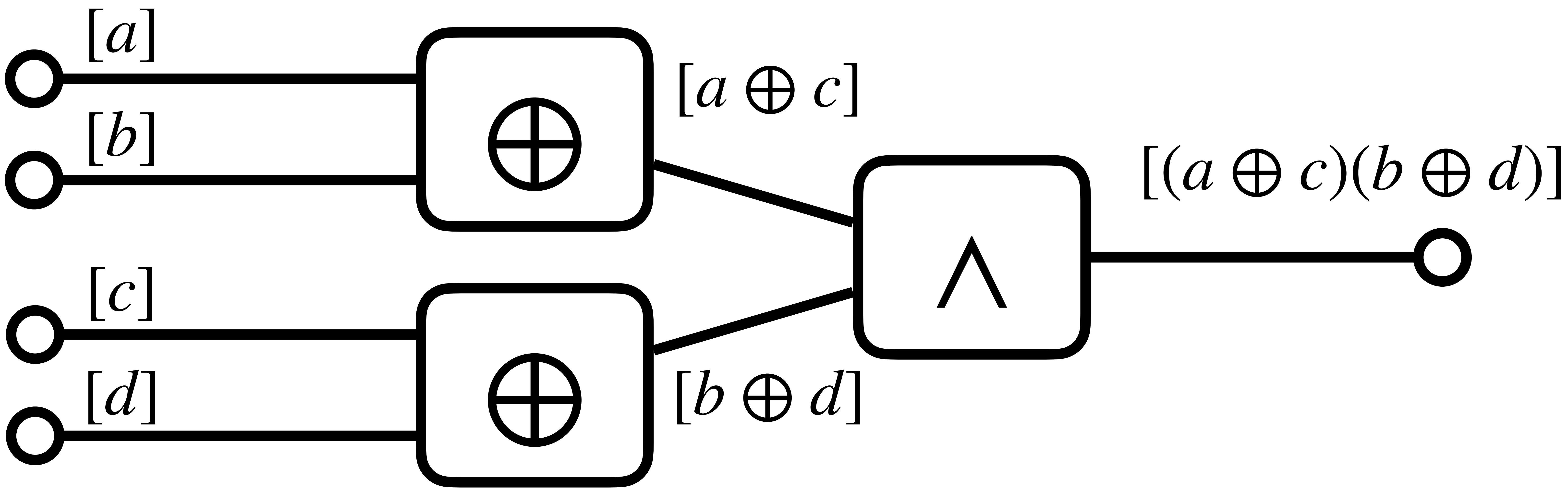
Where do input shares come from?

How do we XOR two shares?

How do we AND two shares?

How do we “decrypt” output shares?





Real World Protocol

Walk gate by gate through circuit, maintaining wire shares

For each input (owned by this party), sample and send shares

For each other input, receive a share

For each XOR, XOR shares

For each AND, sample a bit and call OT functionality twice

For each output, send/ receive shares

Real World Protocol

Walk gate by gate through circuit, maintaining wire shares

For each input (owned by this party), sample and send shares

For each other input, receive a share

For each XOR, XOR shares

For each AND, sample a bit and call OT functionality twice

For each output, send/ receive shares

Simulation

Walk gate by gate through circuit, maintaining simulated wire shares

Real World Protocol

Walk gate by gate through circuit, maintaining wire shares

For each input (owned by this party), sample and send shares

For each other input, receive a share

For each XOR, XOR shares

For each AND, sample a bit and call OT functionality twice

For each output, send/ receive shares

Simulation

Walk gate by gate through circuit, maintaining simulated wire shares

For each input (owned by this party), sample random shares

# Real World Protocol	# Simulation
Walk gate by gate through circuit, maintaining wire shares	*Walk gate by gate through circuit, maintaining simulated wire shares*
For each input (owned by this party), sample and send shares	For each input (owned by this party), sample random shares
For each other input, receive a share	For each other input, sample a share
For each XOR, XOR shares	
For each AND, sample a bit and call OT functionality twice	
For each output, send/ receive shares	

Real World Protocol	Simulation
Walk gate by gate through circuit, maintaining wire shares	*Walk gate by gate through circuit, maintaining simulated wire shares*
For each input (owned by this party), sample and send shares	For each input (owned by this party), sample random shares
For each other input, receive a share	For each other input, sample a share
For each XOR, XOR shares	For each XOR, XOR shares
For each AND, sample a bit and call OT functionality twice	
For each output, send/ receive shares	

Real World Protocol	Simulation
Walk gate by gate through circuit, maintaining wire shares	*Walk gate by gate through circuit, maintaining simulated wire shares*
For each input (owned by this party), sample and send shares	For each input (owned by this party), sample random shares
For each other input, receive a share	For each other input, sample a share
For each XOR, XOR shares	For each XOR, XOR shares
For each AND, sample a bit and call OT functionality twice	For each AND, sample a bit and simulate OT receive by a uniform bit
For each output, send/ receive shares	

Real World Protocol	Simulation
Walk gate by gate through circuit, maintaining wire shares	*Walk gate by gate through circuit, maintaining simulated wire shares*
For each input (owned by this party), sample and send shares	For each input (owned by this party), sample random shares
For each other input, receive a share	For each other input, sample a share
For each XOR, XOR shares	For each XOR, XOR shares
For each AND, sample a bit and call OT functionality twice	For each AND, sample a bit and simulate OT receive by a uniform bit
For each output, send/ receive shares	For each output, compute message consistent with the output

Today's objectives

Discuss randomized functionalities

Update definition of semi-honest security

See a proof of *insecurity*

Consider security proof for GMW protocol