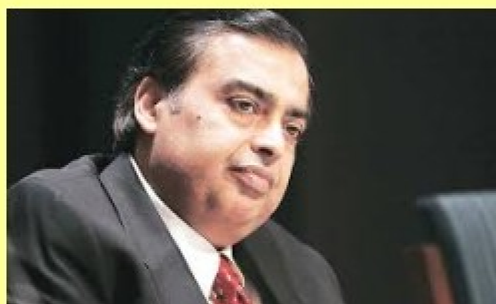
Yao's Millionaires' Problem

Protocols for Secure Computations (Extended Abstract). FOCS 1982: 160-164

Turing award winner Andrew Yao

Yao's millionaires' problem

₹ X



?

<

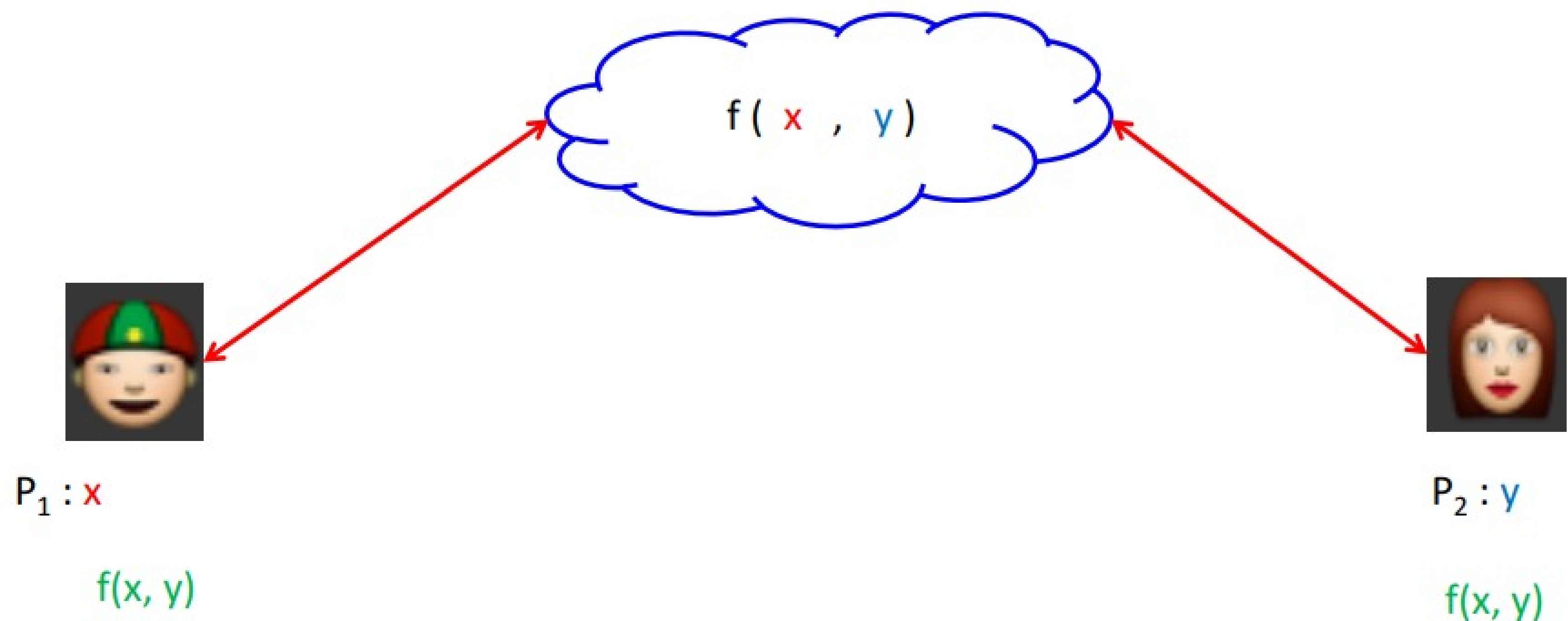
=

>

₹ Y

Find the richer without disclosing exact value of individual assets

Secure 2-PC

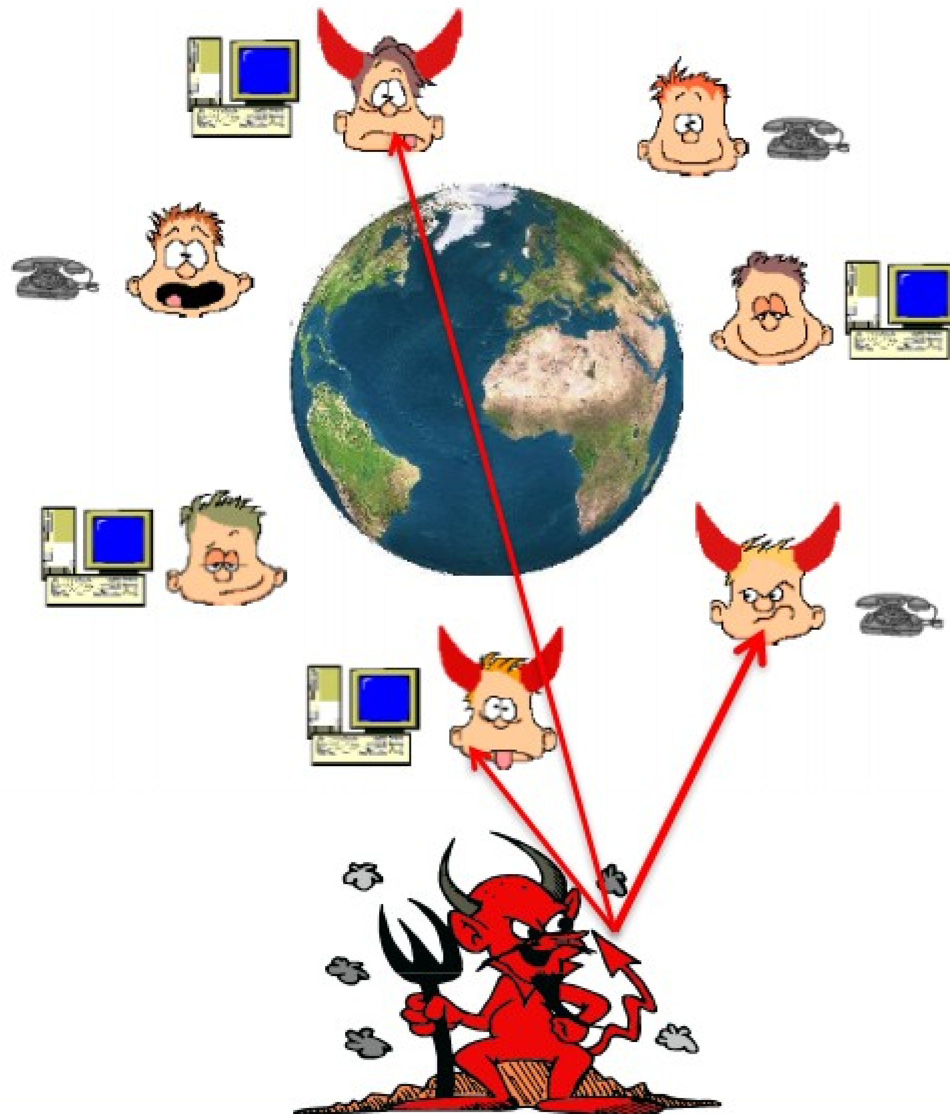


- Mutually distrusting entities with individual private data

- Want to compute a joint function of their inputs without revealing anything beyond

Secure Multiparty Computation (MPC)

MPC – holy grail



Setup:

- n parties $P_1,\ldots,P_n$; 'some' are corrupted
- P_i has private input x_i
- A common n-input function f

Goals:

- **Correctness:** Compute $f(x_1,x_2,..x_n)$
- **Privacy:** Nothing beyond function output must be leaked

Applications: (Dual need of data privacy & data usability)

Preventing Satellite Collision

E-auction

Data Analytics

Privacy-preserving ML

Outsourcing

E-voting

Application of 2PC- Privacy-preserving Data mining

- How many patients suffering from AIDS in total ?
- Are there any common patient registered for disease X in all the hospitals ?
- Varieties of other statistics …

How to solve 2PC?

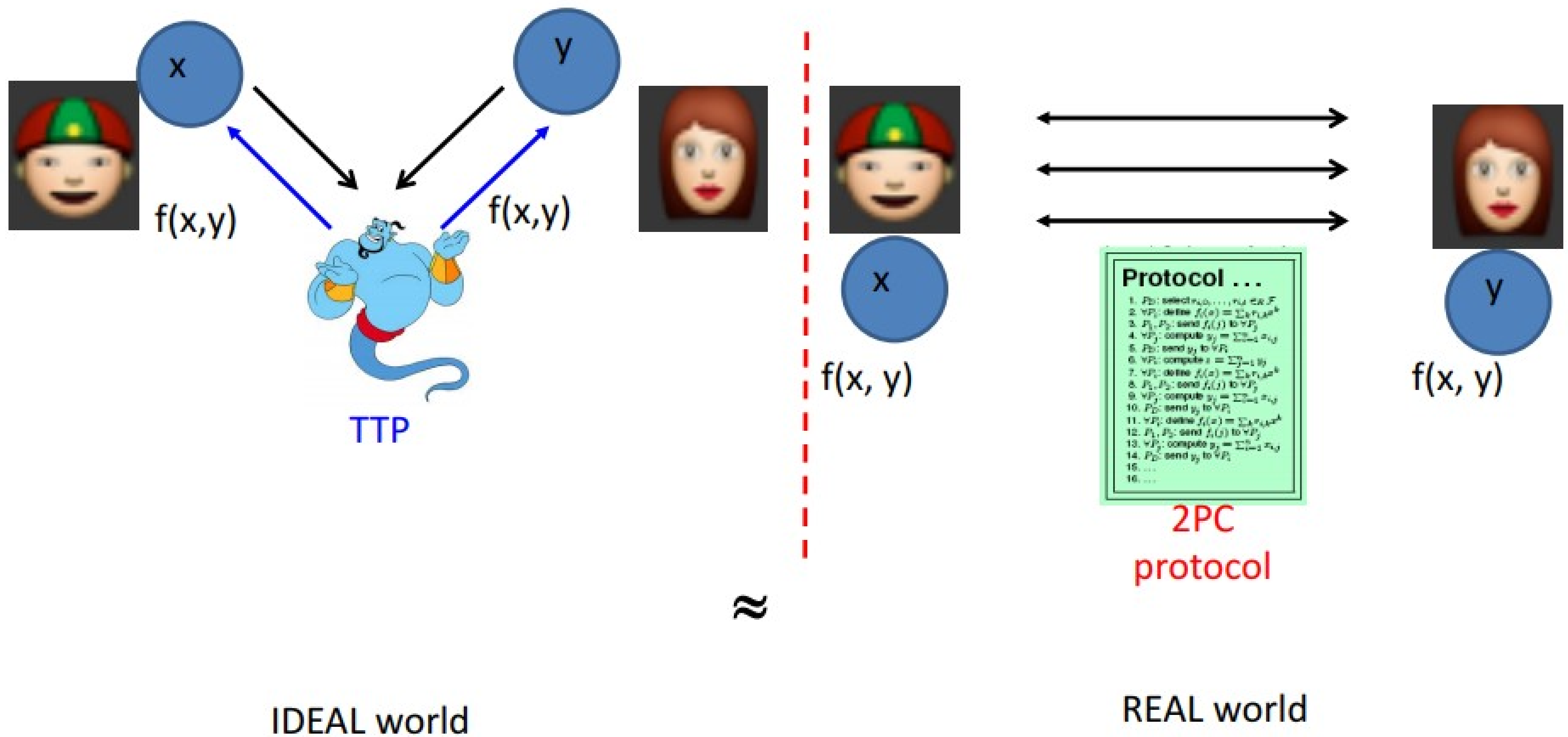
- Trusted third party (TTP) $\rightarrow$ solution for secure 2PC
  - ➢ Send input to TTP, obtain function output : Ideal solution

IDEAL world secure 2PC protocol

TTPs exist only in fairy tales!!

Security goal of 2PC

- Goal of a secure 2PC protocol : emulate the role of a TTP
  - ➢ De-centralizing the trust



Yao's Protocol

- **A protocol for general secure two-party computation**
  - Constant number of rounds
  - The basic protocol is secure only for semi-honest adversaries
  - Many applications of the methodology beyond secure computation
- **General secure computation**
  - Can securely compute *any* functionality
  - Based on a representation of the functionality as a **Boolean circuit**

Representing functions as Boolean circuits???

- In some cases the circuits are small
  - Adding numbers
  - Comparing numbers
  - Multiplying numbers?
  - Computing AES?
  - Working with indirect addressing (A[i]) ?

- We can efficiently do secure computation of millions and billions of gates

Basic ideas

- **A *plain* circuit is evaluated by**
  - Setting values to its input gates
  - For each gate, computing the value of the outgoing wire as a function of the wires going into the gate
- **Secure computation:**
  - No party should learn the values of any internal wires
- **Yao's protocol**
  - A compiler which takes a circuit and transforms it to a circuit which hides all information but the final output

Outline

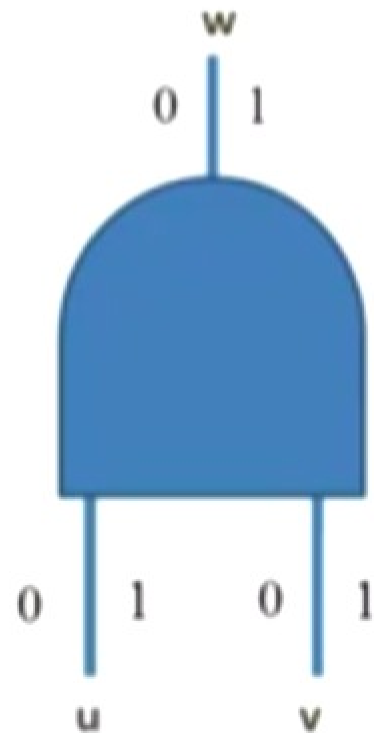
- **Garbled circuit**
  - An encrypted circuit together with a pair of keys (k_0,k_1) for every wire so that for any gate, given one key on every input wire:
    - It is possible to compute the key of the corresponding gate output
    - It is impossible to learn anything else
- **Tool: oblivious transfer**
  - Input: sender has x_0,x_1; receiver has b
  - Receiver obtains x_b only
  - Sender learns nothing

A Garbled Circuit

- For the entire circuit, assign independent random values/keys to each wire (key k_0 for 0, key k_1 for 1)
  - These keys are also called "garbled values"

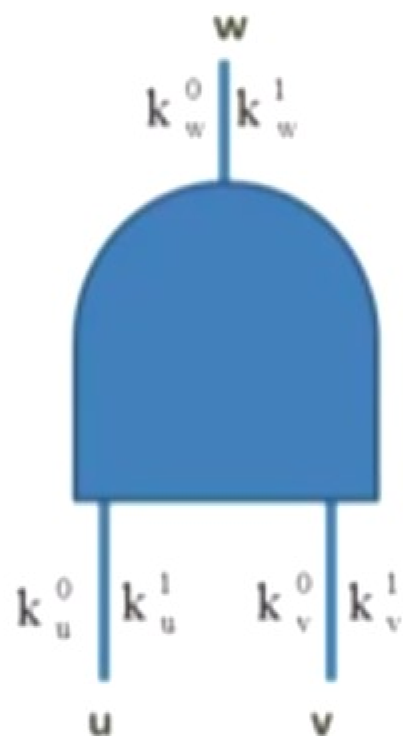
- Encrypt each gate, so that given one key for each input wire, can compute the appropriate key on the output wire

An AND Gate



u	v	w
0	0	0
0	1	0
1	0	0
1	1	1

An AND Gate with Garbled Values



u	v	w
k_u^0	k_v^0	k_w^0
k_u^0	k_v^1	k_w^0
k_u^1	k_v^0	k_w^0
k_u^1	k_v^1	k_w^1

A Garbled AND Gate

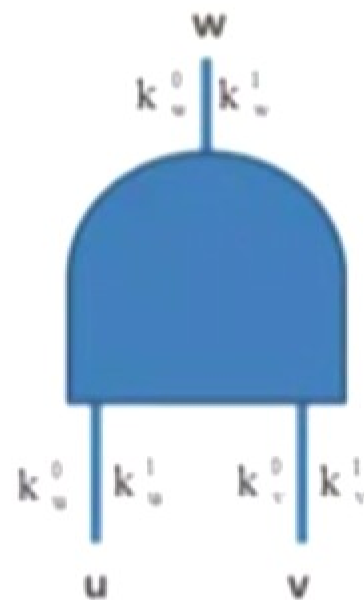
u	v	w
k_u^0	k_v^0	$E_{k_u^0}(E_{k_v^0}(k_w^0))$
k_u^0	k_v^1	$E_{k_u^0}(E_{k_v^1}(k_w^0))$
k_u^1	k_v^0	$E_{k_u^1}(E_{k_v^0}(k_w^0))$
k_u^1	k_v^1	$E_{k_u^1}(E_{k_v^1}(k_w^1))$

A Garbled AND Gate

- **The actual garbled gate**

in permuted order

$$
\left\{
\begin{array}{l}
E_{k_u^1}(E_{k_v^0}(k_w^0)) \\
E_{k_u^0}(E_{k_v^1}(k_w^0)) \\
E_{k_u^1}(E_{k_v^1}(k_w^1)) \\
E_{k_u^0}(E_{k_v^0}(k_w^0))
\end{array}
\right.
$$

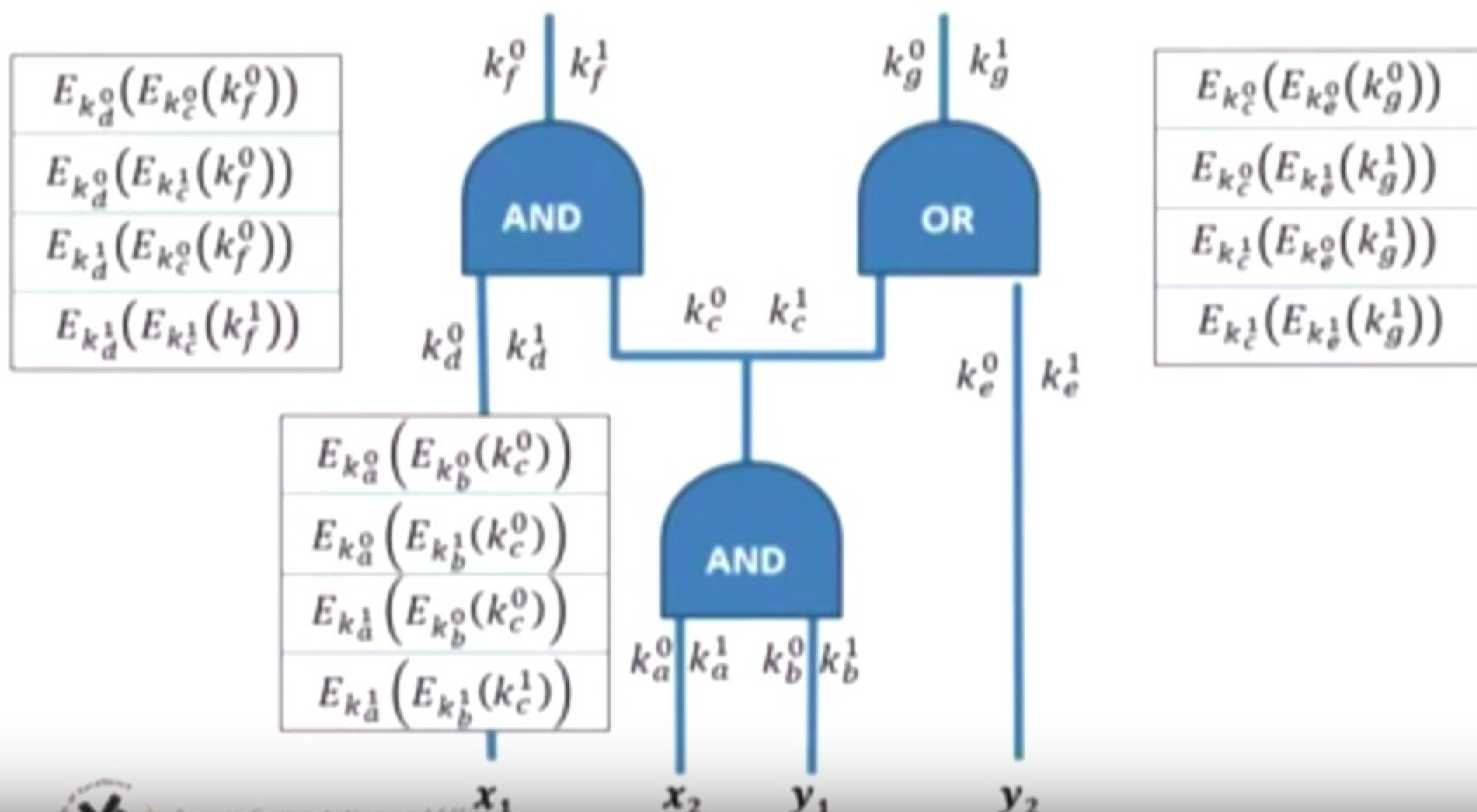


- Given K_u^0 and K_v^1 can obtain only K_w^0
- Furthermore, since the order of the rows is permuted, the party has no idea if it obtained the 0 or 1 key

Constructing a Garbled Circuit

- **Given a Boolean circuit**
  - Assign garbled values to all wires
  - Construct garbled gates using the garbled values
- **Central property:**
  - Given a garbled value for each input wire, can compute the entire circuit, and obtain garbled values for the output wires
  - Given a translation table for the output wires, can obtain output
  - Nothing but the final output is learned!

An Example Circuit



Computing a Garbled Circuit

- **How does the party computing the circuit know that it decrypted the "correct" entry?**
  - A gate table has four entries in permuted order
  - The keys known to the evaluator can decrypt only a single entry, but symmetric encryption may decrypt "correctly" even with incorrect keys
- **Two possibilities** (actually many...)
  - Add redundant zeroes to the plaintext; only correct keys give redundant block
  - Add a bit to signal which ciphertext to decrypt

Computing a Garbled Circuit

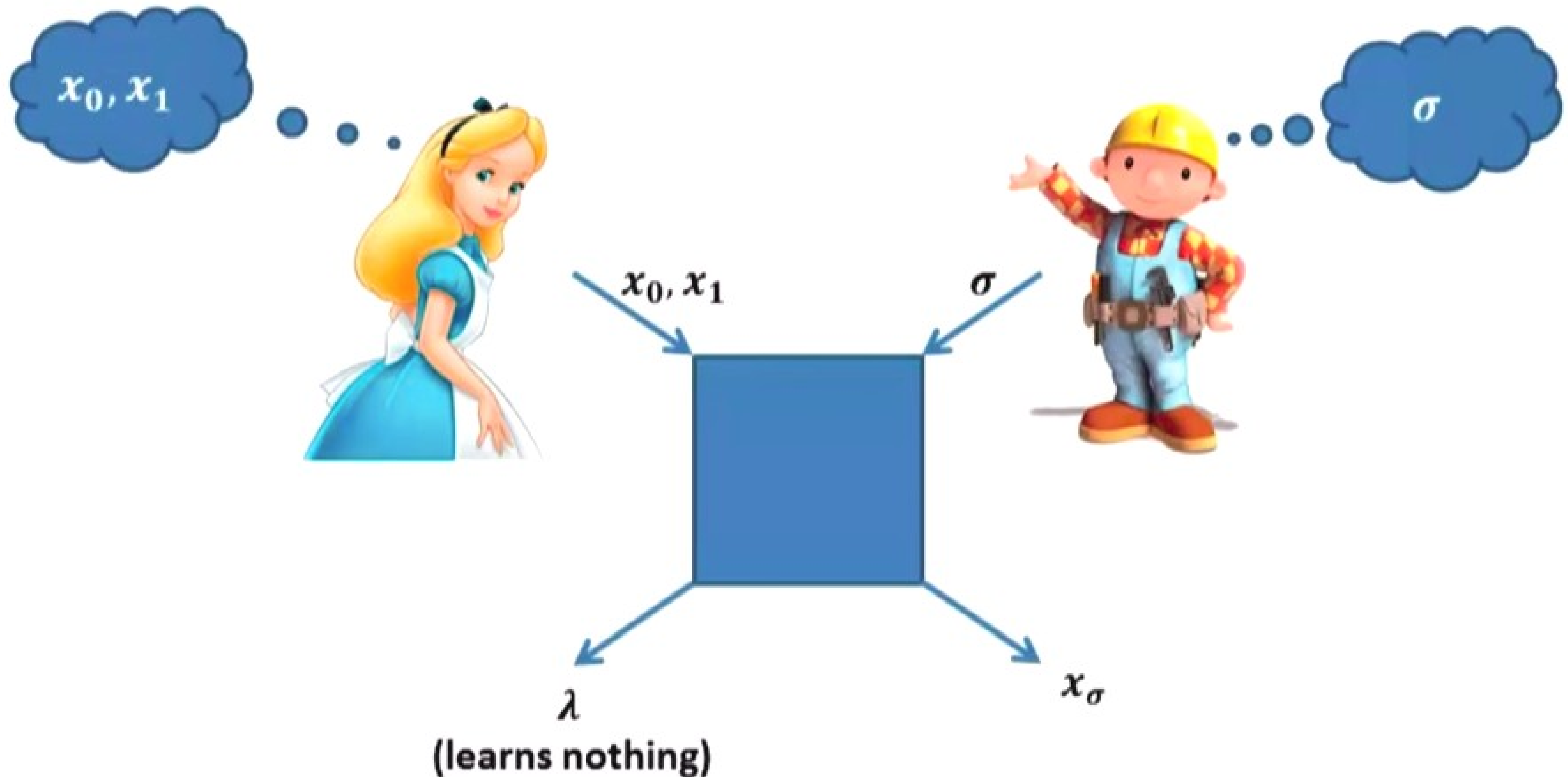
- **Option 1:**
  - Encryption: $\mathbf{E_K(m)} = [\mathbf{r}, \mathbf{F_K(r)} \oplus (\mathbf{m} || \mathbf{0^n})]$
  - By the pseudo-randomness of **F**, the probability of obtaining $\mathbf{0^n}$ with an incorrect **K** is negligible

Yao's protocol

- P_1 sends to P_2
  - Tables encoding each circuit gate.
  - The keys corresponding to P_1's input values.

- If P_2 gets the keys corresponding to its input values, it can compute the output of the circuit, and nothing else.
  - Why can't P_1 provide P_2 with the keys corresponding to both 0 and 1 for P_2's input wires?

Oblivious Transfer (OT)



Called 1-out-of-2 oblivious transfer (OT_1^2)

Efficient OT from DDH

- **Recall the DDH assumption over a group $\mathbb{G}$ of order q with generator g**
  - The DDH assumption says that
    $$\{(g, g^a, g^b, g^{ab})\} \approx \{(g, g^a, g^b, g^c)\}$$
    where $a, b, c \leftarrow \mathbb{Z}_q$ are random

Semi-Honest OT

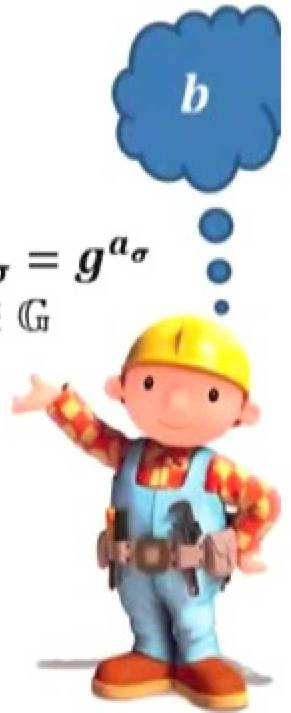
- **Recall ElGamal encryption**
  - **Secret key**: random $a \leftarrow \mathbb{Z}_q$
  - **Public key**: $h = g^a$
  - **Encrypt** $\boldsymbol{m} \in \mathbb{G}$: $c = (u, v) = (g^r, h^r \cdot m)$, random $r \in \mathbb{Z}_q$
  - **Decrypt** $(\boldsymbol{u}, \boldsymbol{v})$: compute $m = \frac{v}{u^a}$
    - Note: $\frac{v}{u^a} = \frac{h^r \cdot m}{(g^r)^a} = \frac{h^r \cdot m}{(g^a)^r} = \frac{h^r \cdot m}{h^r} = m$

Semi-Honest OT

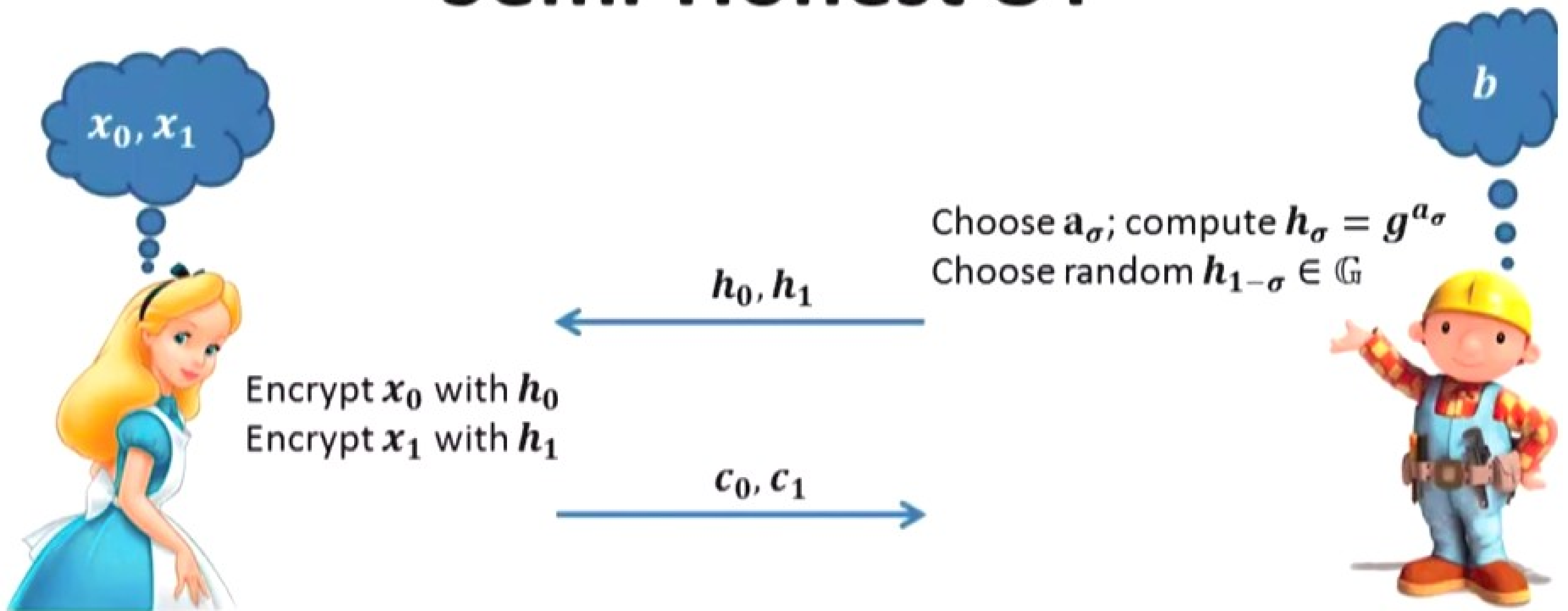
Semi-Honest OT

Choose $\mathbf{a}_\sigma$; compute $\boldsymbol{h_\sigma = g^{a_\sigma}}$

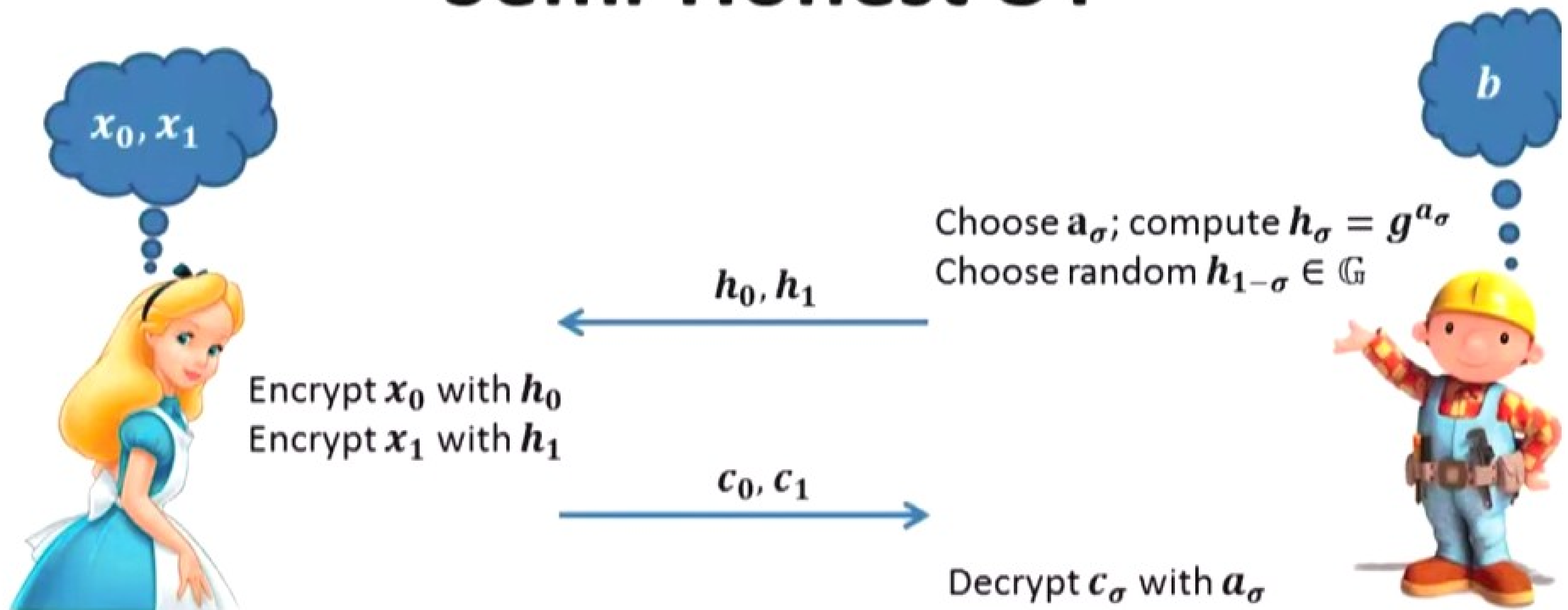
Choose random $\boldsymbol{h_{1-\sigma}} \in \mathbb{G}$



Semi-Honest OT



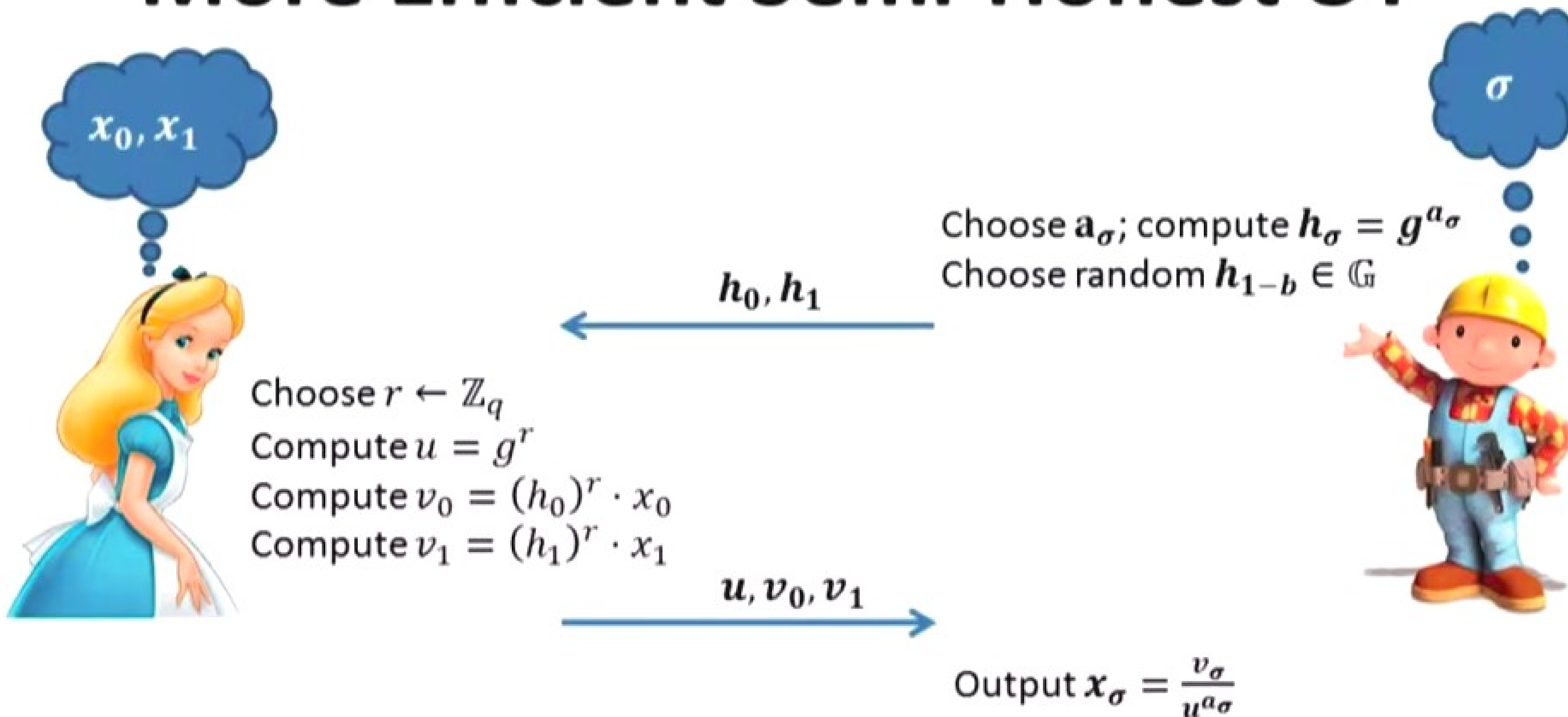
Semi-Honest OT



Note:

- Encrypt x_0 with h_0: $(u_0, v_0) = (g^r, (h_0)^r \cdot x_0)$
- Encrypt x_1 with h_1: $(u_1, v_1) = (g^s, (h_1)^s \cdot x_1)$

More Efficient Semi-Honest OT



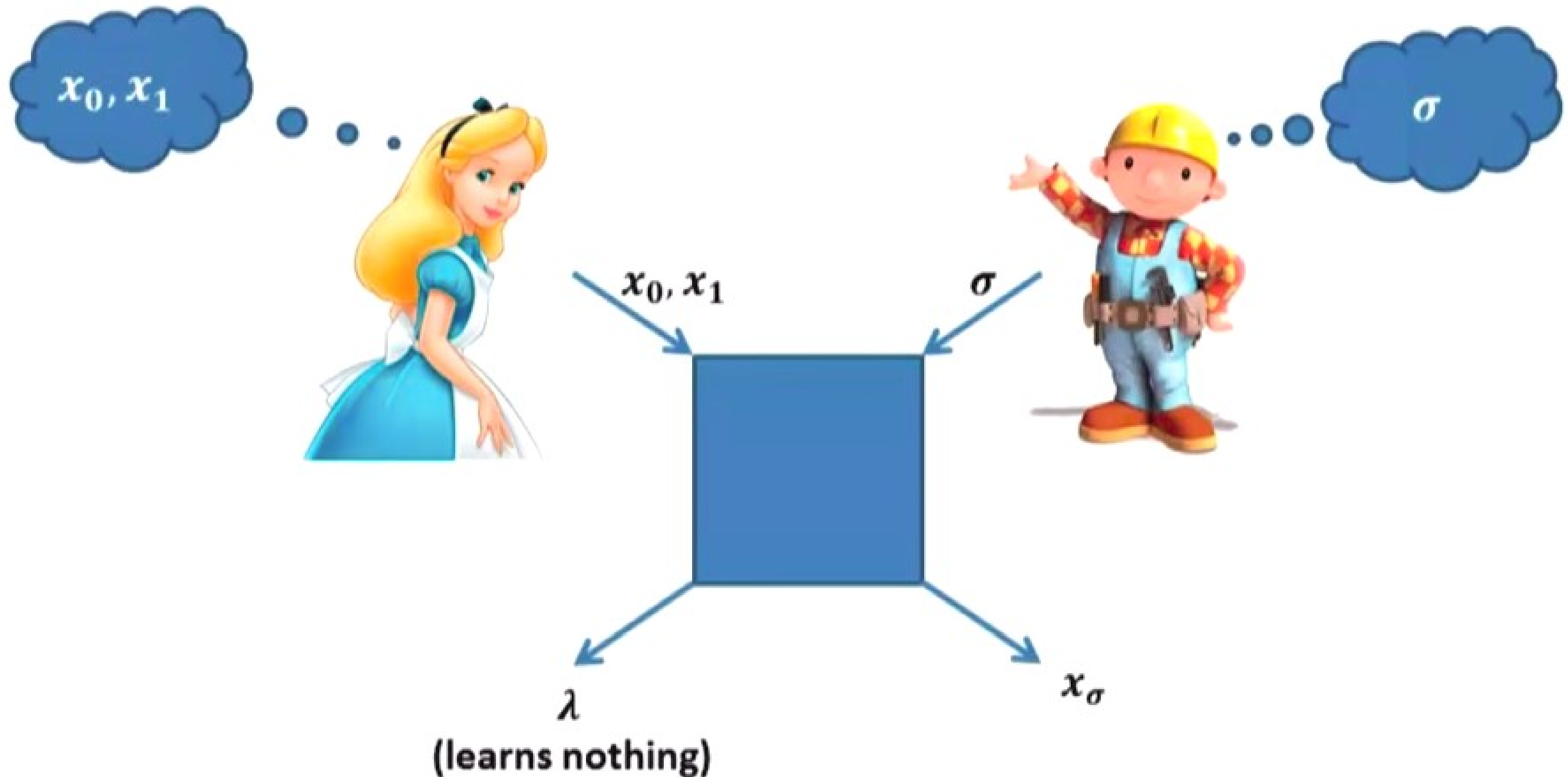
Malicious Adversaries

- **Corrupted sender:**
  - Sender cannot cheat
  - Simulator can "extract" both x_0, x_1 by choosing both h_0 and h_1 so that it knows the secret keys
- **Corrupted receiver:**
  - Receiver can choose both h_0 and h_1 so that it knows the secret keys

Preventing Malicious

- **The idea:**
  - Alice sends a random group element w
  - Bob chooses h_0, h_1 so that $h_0 \cdot h_1 = w$
    - Bob can easily do this by choosing a_σ, computing $h_\sigma = g^{a_\sigma}$ and setting $h_{1-\sigma} = w/h_\sigma$
    - Bob cannot know both DLOGs of h_0, h_1 or it can compute the DLOG of H

Oblivious Transfer (OT)



Called 1-out-of-2 oblivious transfer (OT_1^2)

Yao's protocol

- For every wire i of P_2's input:
  - The parties run an OT protocol
  - P_2's input is her input bit (y_i).
  - P_1's input is k_i^0, k_i^1
  - P_2 learns $k_i^{y_i}$
- The OTs for all input wires can be run in parallel.
- Afterwards P_1 can compute the circuit by itself.

Yao's Protocol

- **Input: x** and **y** of length **n**
- P_1 generates a garbled circuit **G**(**C**)
  - $\mathbf{k_L}^0, \mathbf{k_L}^1$ are the keys on wire $\mathbf{w_L}$
  - Let $w_1, \ldots, w_n$ be the input wires of P_1 and $w_{n+1}, \ldots, w_{2n}$ be the input wires of P_2
- P_1 sends to P_2 **G(C)** and the strings $\mathbf{k_1}^{x1}, \ldots, \mathbf{k_n}^{xn}$
- P_1 and P_2 run **n** OTs in parallel
  - P_1 inputs $(\mathbf{k_{n+i}}^0, \mathbf{k_{n+i}}^1)$
  - P_2 inputs $\mathbf{y_i}$
- Given all keys, P_2 computes **G**(**C**) and obtains **C**(**x,y**)
  - P_2 sends result to P_1

The Example Circuit

(input wires P_1= d,a; P_2 = b,e)