CS4501 Cryptographic Protocols
Lecture 11: 🦫🦫🦫, The Limits of Perfect MPC

https://jackdoerner.net/teaching/#2026/Spring/CS4501

Essential vs Convenience Gates:

1. When we were discussing arithmetic circuits, we argued that it is necessary only to have $+$, $\times$,in, and out gates.

2. When I actually described the BGW protocol in **Lecture 8**, I added a rand gate to make it easy to express randomized functions. This gate is not strictly necessary, as you will see in your homework, but it is convenient.

3. From now on, I will mostly use only the *necessary* gates to talk about costs and feasibility.

Essential vs Convenience Gates:

4. On the other hand, when talking about constructing circuits, I will assume we have two additional *convenience gates* (besides rand). Both of them can be constructed using $+, \times$, and in in the semi-honest setting, and both can also be constructed *directly*, in a way that avoids communication.

- The scalar gate $(\mathsf{scale}, c, i, o)$ that sets the value of wire o to $c \cdot w_i$ where w_i is the value on wire i and $c \in \mathbb{F}_p$ is a constant. You saw the technique for constructing this gate without communication in **Lecture 7.**

- The constant gate (const, c, o) sets the value of wire o to $c \in \mathbb{F}_p$. *How can you construct this gate without communication?*

  Answer: If every party sets the value its share to c, then they have a degree-0 polynomial (i.e. a flat line) encoding the "secret" c.

Recap: The BGW Protocol $\pi_{\mathsf{BGW}}(n, t, p, C)$

Let $t < n < p \in \mathbb{N}$. There are n parties $P_1, \ldots, P_n$, with inputs $x_1, \ldots, x_n \in \mathbb{F}_p$ respectively. π_{BGW} computes a well-formed n-ary arithmetic circuit $C : \mathbb{F}_p^n \to \mathbb{F}_p^n$. The parties output $y_1, \ldots, y_n \in \mathbb{F}_p$ respectively.

There are Three Phases:

1. **Input Sharing:** every P_i with input x_i finds the entry $(\mathsf{in}, i, o) \in C$, computes $\langle w_o \rangle \leftarrow \mathsf{Share}_{p,n,t}(x_i)$ and sends $\langle w_o \rangle_j$ to every P_j for $j \in [n] \backslash \{i\}$.

2. **Circuit Eval:** the parties traverse the circuit C in topological order, jointly evaluating each gate, using shares of its input wires to produce shares of its output wire.

   - Suppose the parties arrive at gate $(+, j, k, o) \in C$. Each party P_i individually computes $\langle w_o \rangle_i := \langle w_j \rangle_i + \langle w_k \rangle_i$.

Recap: The BGW Protocol $\pi_{\mathsf{BGW}}(n, t, p, C)$

Let $t < n < p \in \mathbb{N}$. There are n parties $P_1, \ldots, P_n$, with inputs $x_1, \ldots, x_n \in \mathbb{F}_p$ respectively. π_{BGW} computes a well-formed n-ary arithmetic circuit $C : \mathbb{F}_p^n \to \mathbb{F}_p^n$. The parties output $y_1, \ldots, y_n \in \mathbb{F}_p$ respectively.

There are Three Phases:

2. **Circuit Eval:** the parties traverse the circuit C in topological order, jointly evaluating each gate, using shares of its input wires to produce shares of its output wire.

   - Suppose the parties arrive at gate $(+, j, k, o) \in C$. Each party P_i individually computes $\langle w_o \rangle_i := \langle w_j \rangle_i + \langle w_k \rangle_i$.
   - Suppose the parties arrive at gate $(\times, j, k, o) \in C$. Each party P_i sends $\left(\langle w_j \rangle_i, \langle w_k \rangle_i\right)$ to $\mathcal{F}_{\mathsf{mul}}$ and receives $\langle w_o \rangle_i$ in response.

Recap: The BGW Protocol $\pi_{\mathsf{BGW}}(n, t, p, C)$

Let $t < n < p \in \mathbb{N}$. There are n parties $P_1, \ldots, P_n$, with inputs $x_1, \ldots, x_n \in \mathbb{F}_p$ respectively. π_{BGW} computes a well-formed n-ary arithmetic circuit $C : \mathbb{F}_p^n \to \mathbb{F}_p^n$. The parties output $y_1, \ldots, y_n \in \mathbb{F}_p$ respectively.

There are Three Phases:

- Suppose the parties arrive at gate $(+, j, k, o) \in C$. Each party P_i individually computes $\langle w_o \rangle_i := \langle w_j \rangle_i + \langle w_k \rangle_i$.
- Suppose the parties arrive at gate $(\times, j, k, o) \in C$. Each party P_i sends $\left(\langle w_j \rangle_i, \langle w_k \rangle_i\right)$ to $\mathcal{F}_{\mathsf{mul}}$ and receives $\langle w_o \rangle_i$ in response.

3. **Output Reconstruction:** Each P_i finds every output wire $(\mathsf{out}, k, j) \in C$ and sends $\langle w_j \rangle_i$ to P_k. P_k receives $\langle w_j \rangle$, computes $y_k := \mathsf{Recon}_{p,n,t}([n], \langle w_j \rangle)$, and outputs y_k.

Recap: The BGW Multiplication Protocol

Inputs: Each P_i begins with $\langle w \rangle_i$ and $\langle w' \rangle_i$.

1. Without interacting, every P_i computes $\widehat{\langle w \cdot w' \rangle}_i = \langle w \rangle_i \cdot \langle w' \rangle_i$.
2. Every P_i samples $\langle 0_i \rangle \leftarrow \mathsf{Share}_{p,n,2t}(0)$ and sends $\langle 0_i \rangle_j$ to every P_j for $j \in [n] \backslash \{i\}$.
3. Every P_i computes $\widetilde{\langle w \cdot w' \rangle}_i = \widehat{\langle w \cdot w' \rangle}_i + \sum_{k \in [n]} \langle 0_k \rangle_i$.
4. The parties invoke $\mathscr{F}_{\mathsf{SFE}}(n, f_{\mathsf{reduce}}, \mathbb{F}_p, \ldots, \mathbb{F}_p)$ where $f_{\mathsf{reduce}}(\vec{x}) = V_{[n]}\, H_{n,t}\, V_{[n]}^{-1}\, \vec{x}$. Each P_i supplies $\widetilde{\langle w \cdot w' \rangle}_i$ as its input and receives $\langle w \cdot w' \rangle_i$ as its output.
5. Because f_{reduce} is *linear*, we can realize $\mathscr{F}_{\mathsf{SFE}}(n, f_{\mathsf{reduce}}, \mathbb{F}_p, \ldots, \mathbb{F}_p)$ using the BGW protocol *without* any multiplication gates.

Outputs: Each P_i ends with $\langle w \cdot w' \rangle_i$.

Total bandwidth cost: n inputs $+$ n outputs $+$ n zero-sharings $= 3n^2 |p|$.
Total Rounds: 3.

Recap: $\pi_{\mathsf{GRR}}(n, t, p)$.

Inputs: Every P_i for $i \in [n]$ has input $\langle w \rangle_i = f(i)$ where $f \in \mathscr{P}_{p,t,w}$ and input $\langle w' \rangle_i = f'(i)$ where $f' \in \mathscr{P}_{p,t,w'}$.

1. Every P_i locally computes $\hat{g}(i) = f(i) \cdot f'(i) = \langle w \rangle_i \cdot \langle w' \rangle_i$. Note that $\hat{g} \in \mathscr{P}_{p,2t,w\cdot w'}$.
2. Every P_i samples $\langle \hat{g}(i) \rangle \leftarrow \mathsf{Share}_{p,n,t}(\hat{g}(i))$ and sends $\langle \hat{g}(i) \rangle_j$ to P_j for $j \in [n] \backslash \{i\}$.
3. Every P_i computes $\langle w \cdot w' \rangle_i = \sum_{j\in[n]} \ell_j(0) \cdot \langle \hat{g}(j) \rangle_i$.

Outputs: Each P_i ends with $\langle w \cdot w' \rangle_i$.

Total bandwidth cost: $n^2|p|$.
Total Rounds: 1.

BGW+GRR Protocol Bandwidth Costs

- Let C be a circuit over $\mathbb{F}_p$ that we wish to compute.
- Let c_{in} be the number of input gates; i.e. $c_{\mathsf{in}} := |\{(\mathsf{in}, i, o) : (\mathsf{in}, i, o) \in C\}|$.
- Let c_{out} be the number of output gates; i.e. $c_{\mathsf{out}} := |\{(\mathsf{out}, i, j) : (\mathsf{out}, i, j) \in C\}|$.
- Let $c_{\times}$ be the number of multiplication gates; $c_{\times} := |\{(\times, j, k, o) : (\times, j, k, o) \in C\}|$.
- The total bandwidth cost of running the BGW protocol (with GRR multiplication) on C is $c_{\mathsf{total}} = c_{\mathsf{in}} \cdot n \cdot |p| + c_{\mathsf{out}} \cdot n \cdot |p| + c_{\times} \cdot n^2 \cdot |p|$.
- For many circuits, it's clear that the dominating bandwidth cost comes from the number of multiplication gates. *Can we do better?*
  In particular, can we remove the square?

What do We Know About Bandwidth Costs?

In particular, can we remove the square?




- The BGW protocol was published in 1988.
- In 2007 (~20 years later) Damgård and Nielsen introduced an information-theoretically secure protocol that requires $O((c_{\text{in}} + c_{\text{out}} + c_{\times}) \cdot n \cdot |p|)$ bits to be transmitted in the semi-honest setting. A dramatic improvement!
- In 2008, Beerliová-Trubíniová and Hirt improved achieved Perfect Security against malicious adversaries with similar asymptotic costs.
- In 2019 (~10 years later), Damgård proved that there exist circuits such that *any* protocol that perfectly securely computes one of those those circuits must transmit $\Omega(n \cdot c_{\times})$ bits overall (even if the adversary is semi-honest).
- We won't cover those results in this class. We *will* ask something slightly easier.

Motivating Preprocessing

- If we know in *advance* what function we want to compute, but not what the inputs are, can we do some work ahead of time to make the computation more efficient when the inputs arrive?

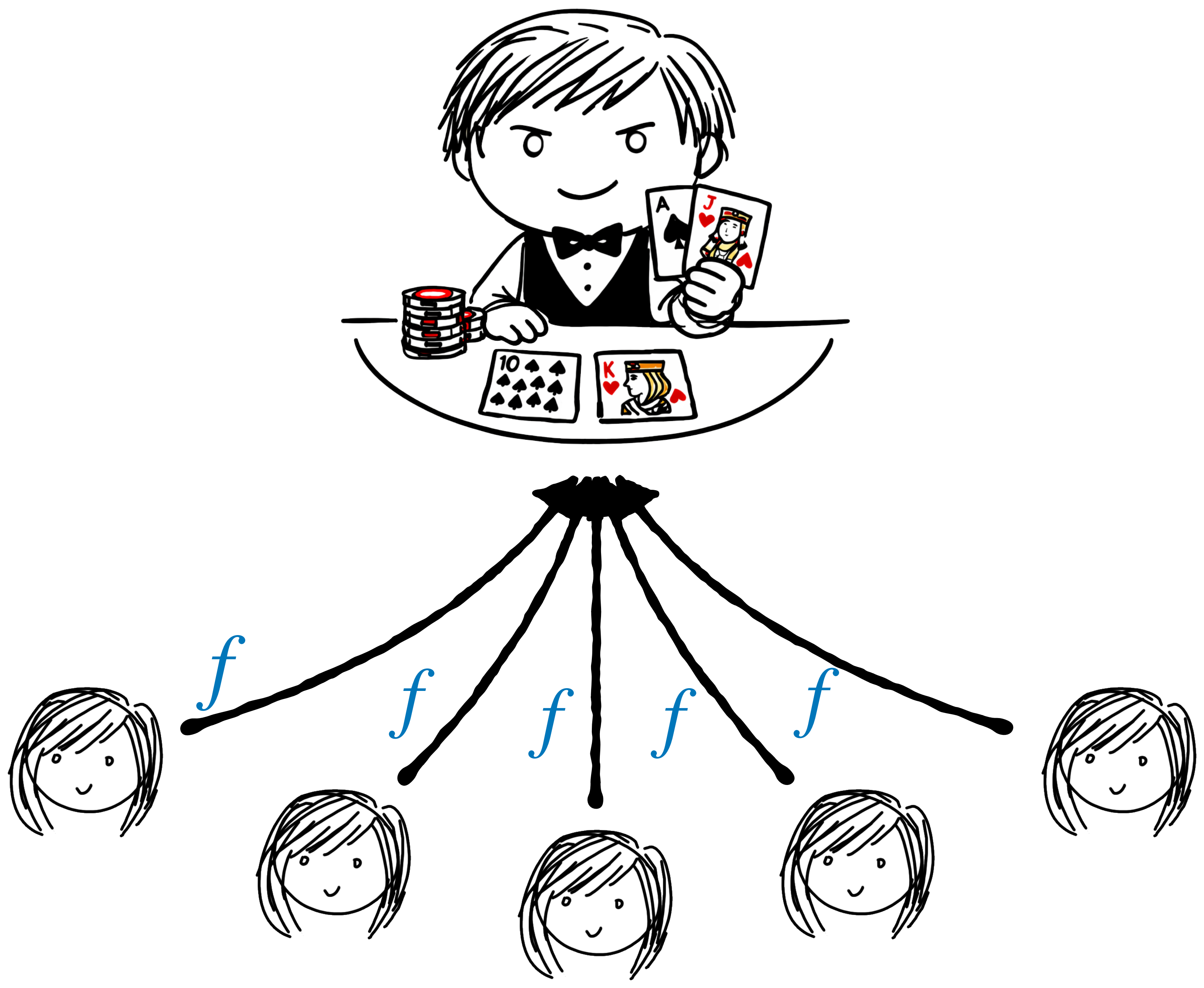
- **Example:** Suppose some hospitals want to run a joint study on their patients. It might take them a while to collect data, but they already know what analysis they will do.

- **Example:** Suppose we want to issue digital credentials *only* when a committee agrees to it. We will create those credentials using a protocol. We know there will be a certain number of credential requests during the day. We would like to work through the night beforehand to make issuing them fast.

The Preprocessing Model

- In cases like these, it is possible to redesign our protocol so that it has two *phases.*

- In the *preprocessing* (or *offline* or *setup*) phase, the parties know the function f that they wish to compute, but not the inputs.



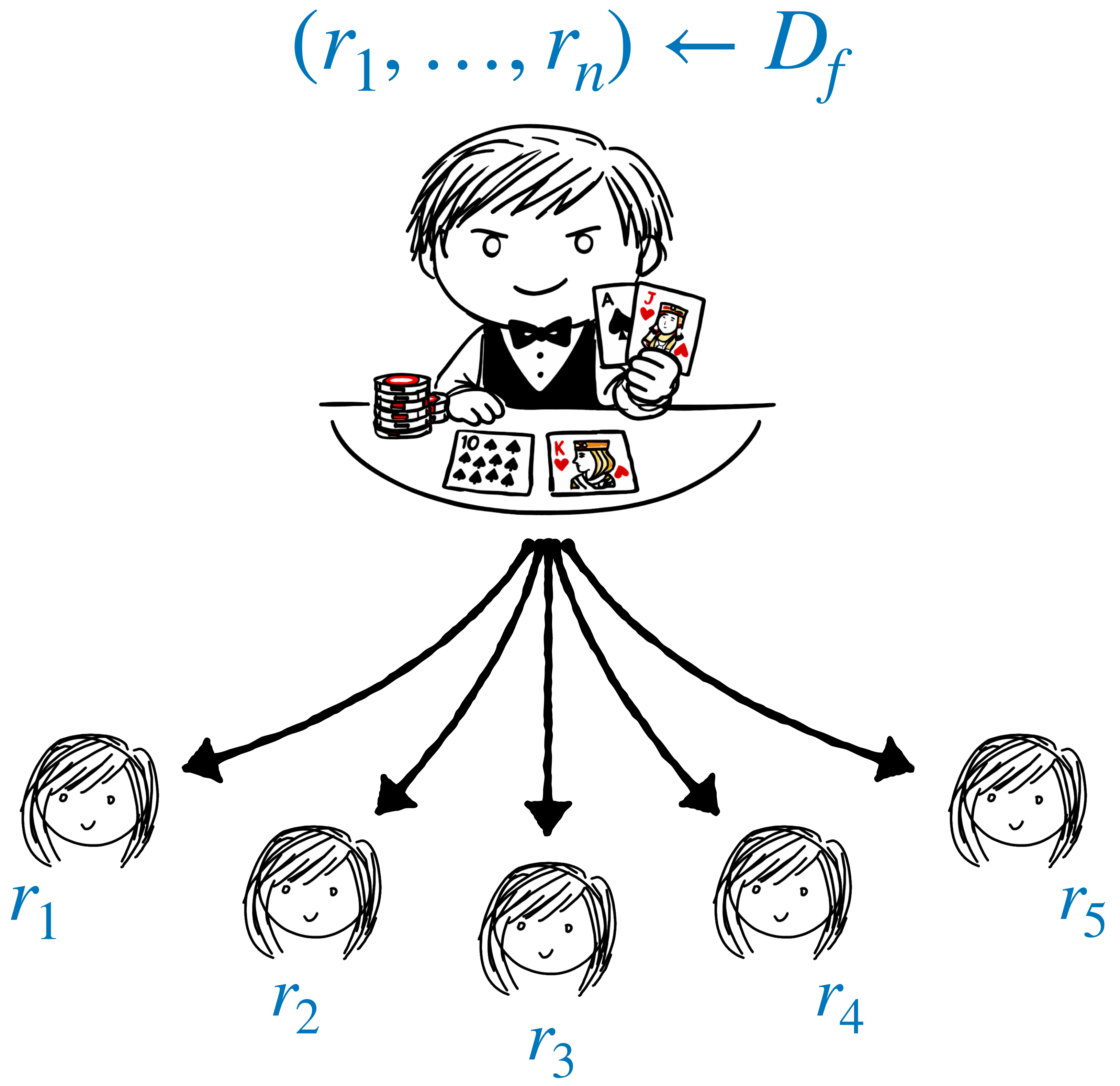
The Preprocessing Model

- In cases like these, it is possible to redesign our protocol so that it has two *phases.*

- In the *preprocessing* (or *offline* or *setup*) phase, the parties know the function f that they wish to compute, but not the inputs.

  A *trusted dealer* samples some *correlated randomness* $(r_1, \ldots, r_n) \leftarrow D_f$ where D_f is some public distribution that depends upon f, and sends r_i to each P_i.

  The r_i values are related to each other in some way. Often the dealer samples $(r_1, \ldots, r_n)$ uniformly from the set of all sets of values that satisfy some constraint.



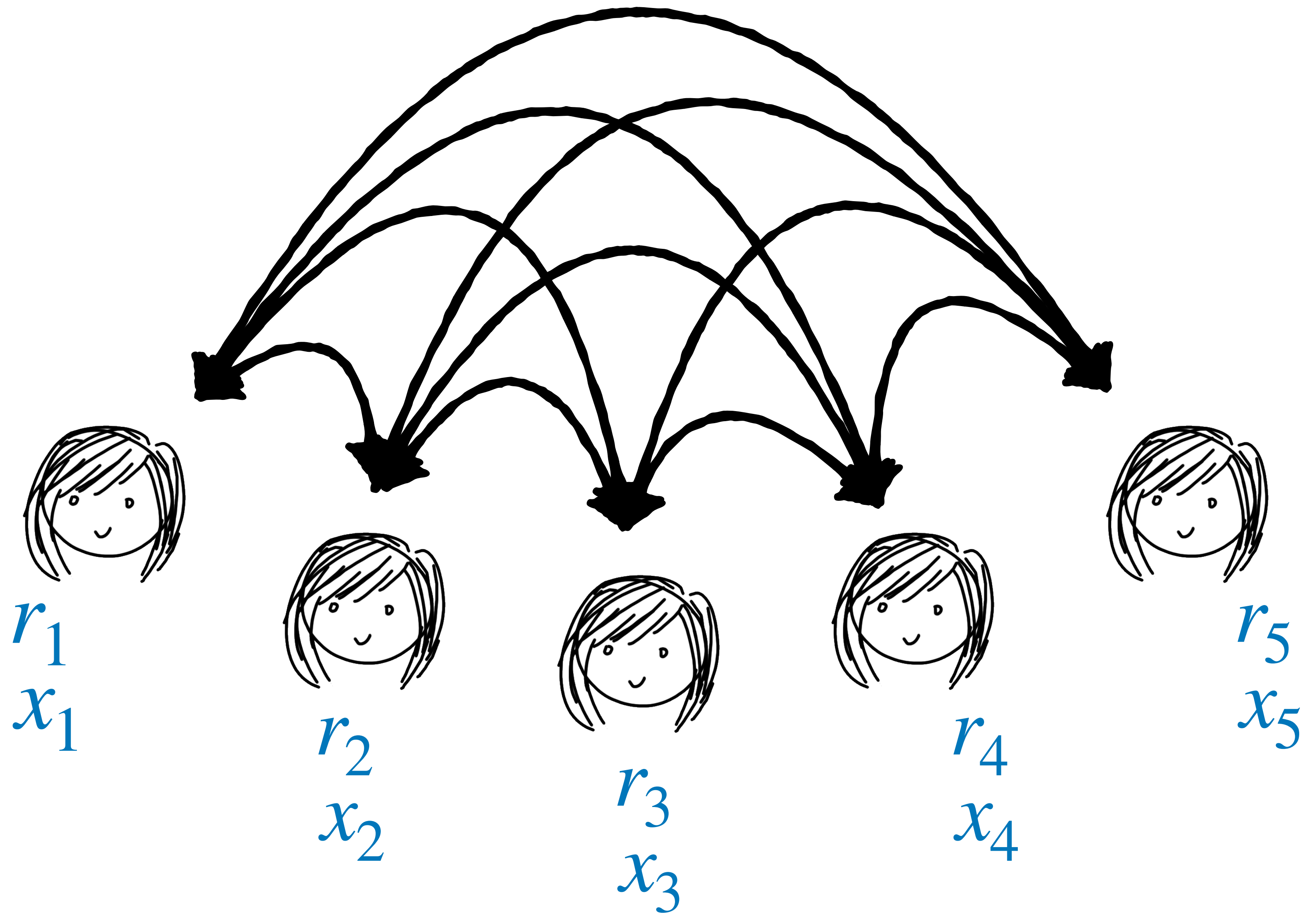
The Preprocessing Model

The r_i values are related to each other in some way. Often the dealer samples $(r_1, \ldots, r_n)$ uniformly from the set of all sets of values that satisfy some constraint.

- In the *online* phase, each P_i receives its input x_i, and uses it together with r_i to securely compute $f(x_1, \ldots, x_n)$. The dealer is not involved.

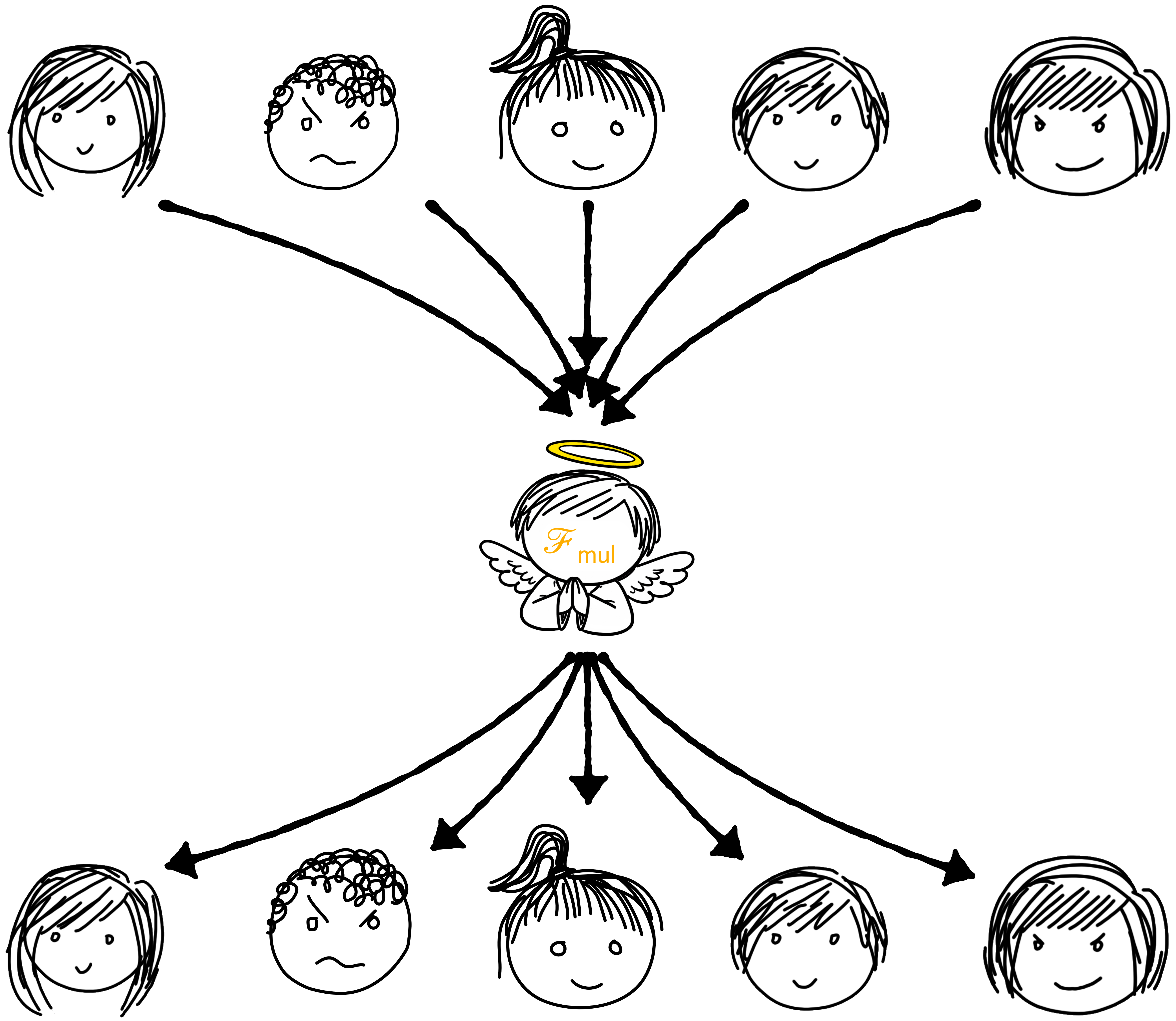
- The offline phase should be more efficient than if we used a one-phase protocol.

- **Question:** *How can we find a dealer? Who can we trust to generate secrets for everyone?*



Reminder: The Ideal Process

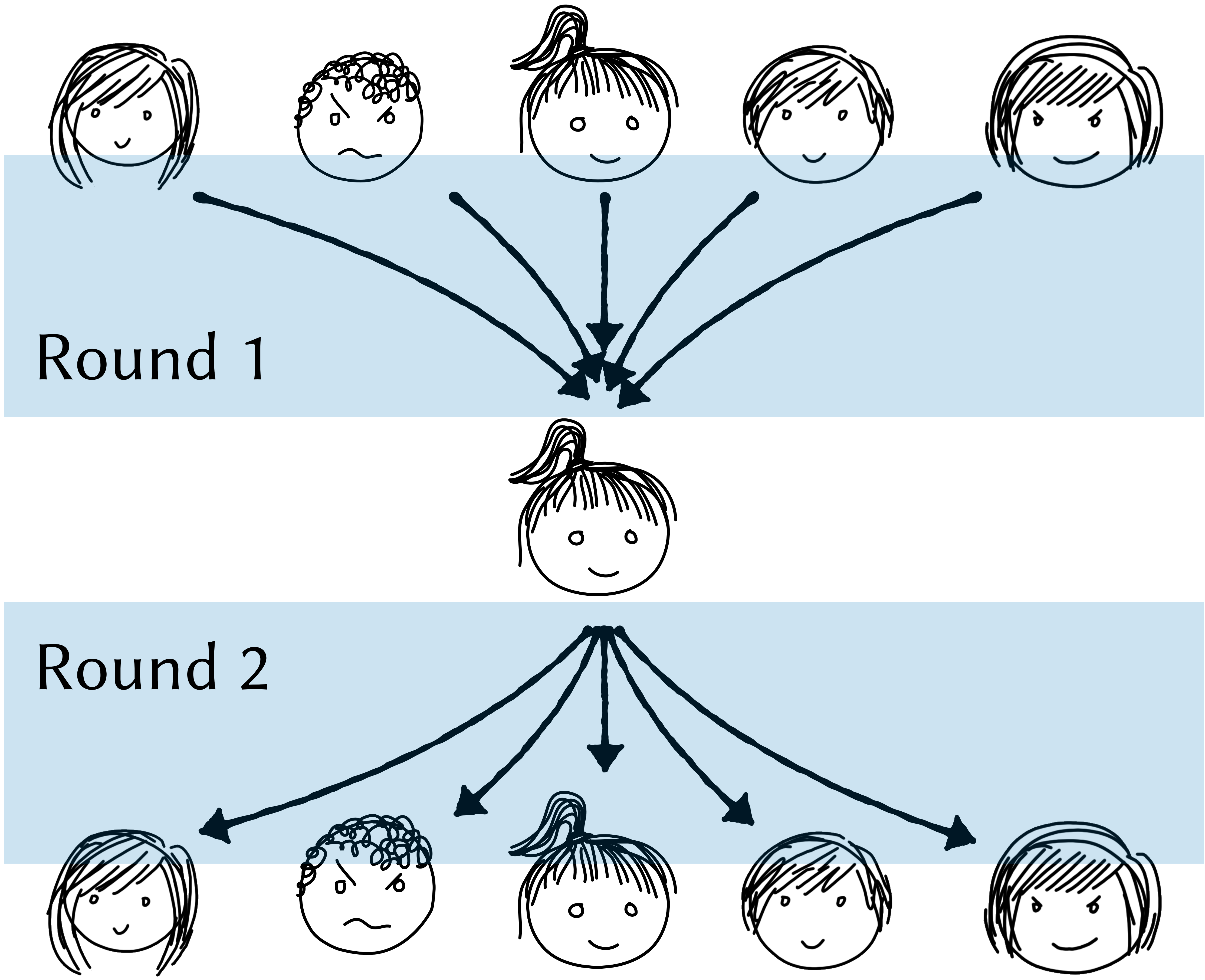
- Each P_i for $i \in [n]$ sends inputs $\langle w_1 \rangle_i$ and $\langle w_2 \rangle_i$ to $\mathscr{F}_{\mathsf{mul}}$.
- $\mathscr{F}_{\mathsf{mul}}$ performs the following steps:
  1. Reconstruct w_1 from $\langle w_1 \rangle$ and w_2 from $\langle w_2 \rangle$ (abort if this fails).
  2. Compute $w_3 := w_1 \cdot w_2$.
  3. Sample $g \leftarrow \mathscr{P}_{p,t,w_3}$ and let $\langle w_3 \rangle := (g(1), \ldots, g(n))$. i.e. let $\langle w_3 \rangle \leftarrow \mathsf{Share}_{p,n,t}(w_3)$.
  4. Send each $\langle w_3 \rangle_i$ to P_i.



Reminder: The Ideal Process

Let's think of this as a protocol:

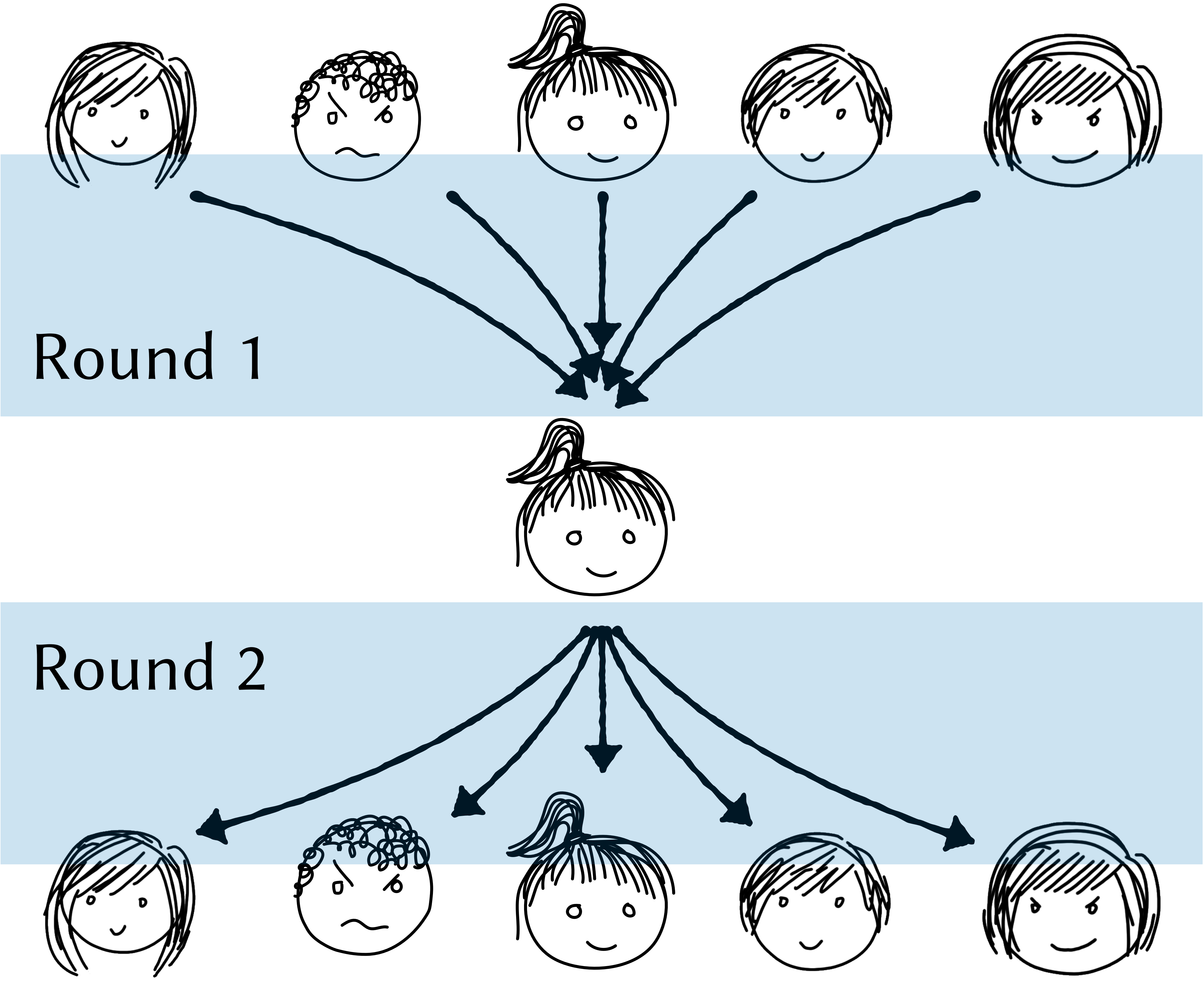
- In round 1, $2n \cdot |p|$ bits are transmitted.
- In round 2, $n \cdot |p|$ bits are transmitted.
- Since we're talking assuming a semi-honest adversary, we can assume that any party could do the job of $\mathscr{F}_{\mathsf{mul}}$ and the protocol would remain *correct*.
- The only problem is that  isn't allowed to learn the product.
- Is there a way for  to compute the product without learning what it is?



Masking Multiplication

Inspiration from OTP:

- Suppose that the parties also had sharings $\langle r_1 \rangle$ and $\langle r_2 \rangle$ such that $r_1 \leftarrow \mathbb{F}_p$ and $r_2 \leftarrow \mathbb{F}_p$ and nobody knows r_1 or r_2.
- The parties can non-interactively compute $\langle v_1 \rangle := \langle w_1 \rangle - \langle r_1 \rangle$ and $\langle v_2 \rangle := \langle w_2 \rangle - \langle r_2 \rangle$, and then each P_i sends its shares to , who reconstructs v_1, v_2.
- v_1, v_2 don't reveal anything about w_1, w_2, but multiplying them yields $v_1 \cdot v_2 = w_1 \cdot w_2 - w_1 \cdot r_2 - w_2 \cdot r_1 + r_1 \cdot r_2$. This approach would require us to securely remove the last 3 terms....

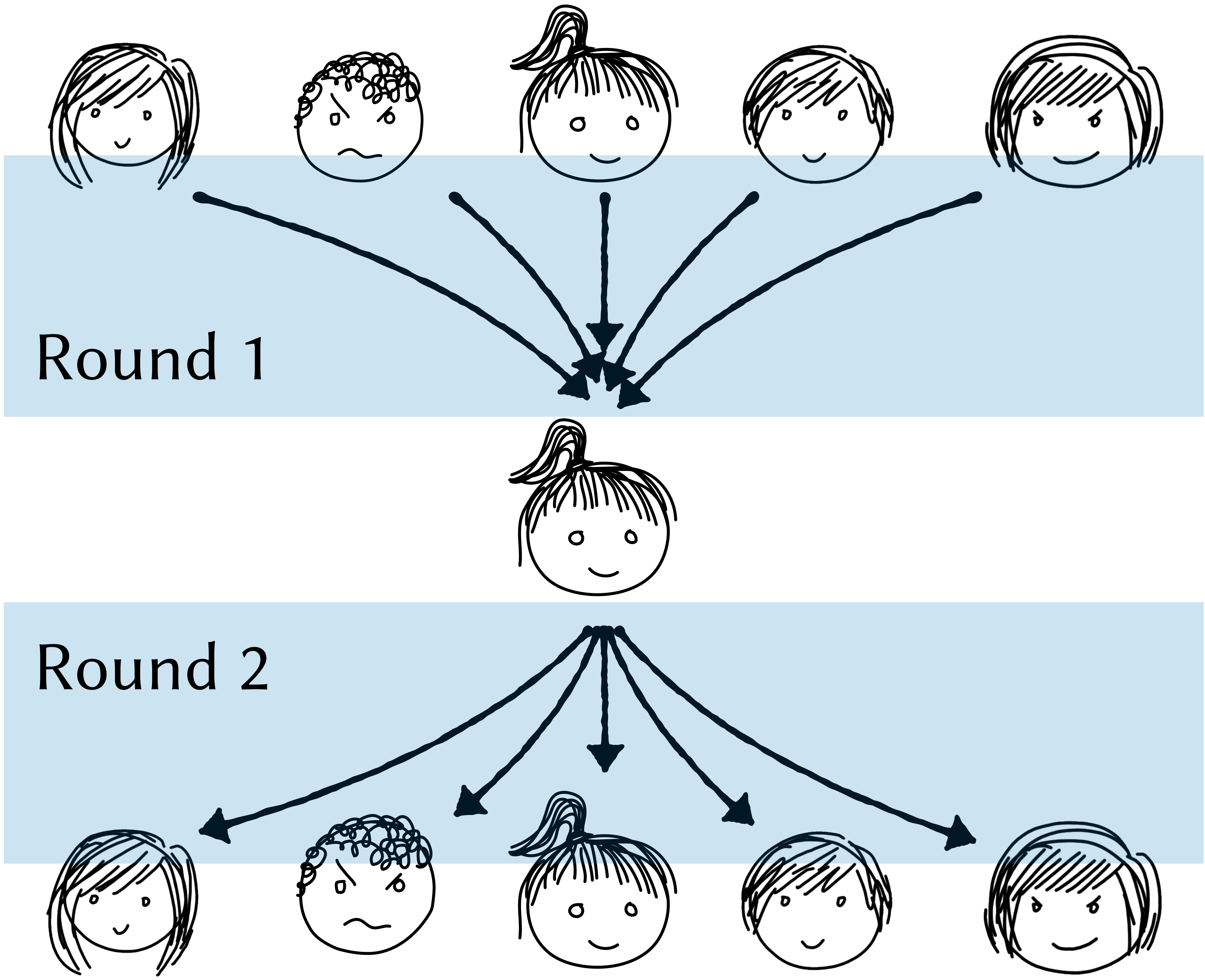


Masking Multiplication

- v_1, v_2 don't reveal anything about w_1, w_2, but multiplying them yields $v_1 \cdot v_2 = w_1 \cdot w_2 - w_1 \cdot r_2 - w_2 \cdot r_1 + r_1 \cdot r_2$. This approach would require us to securely remove the last 3 terms....

Observation 1: the unwanted terms are linear in w_1, w_2, so maybe we can find a way to remove them without interacting?

Observation 2: Since v_1 is *public*, we can compute $v_1 \cdot \langle r_2 \rangle = \langle w_1 \cdot r_2 - r_1 \cdot r_2 \rangle$ using a non-interactive *scalar* gate. This gives us $v_1 \cdot v_2 + v_1 \cdot \langle r_2 \rangle + v_2 \cdot \langle r_1 \rangle = \langle w_1 \cdot w_2 - r_1 \cdot r_2 \rangle$

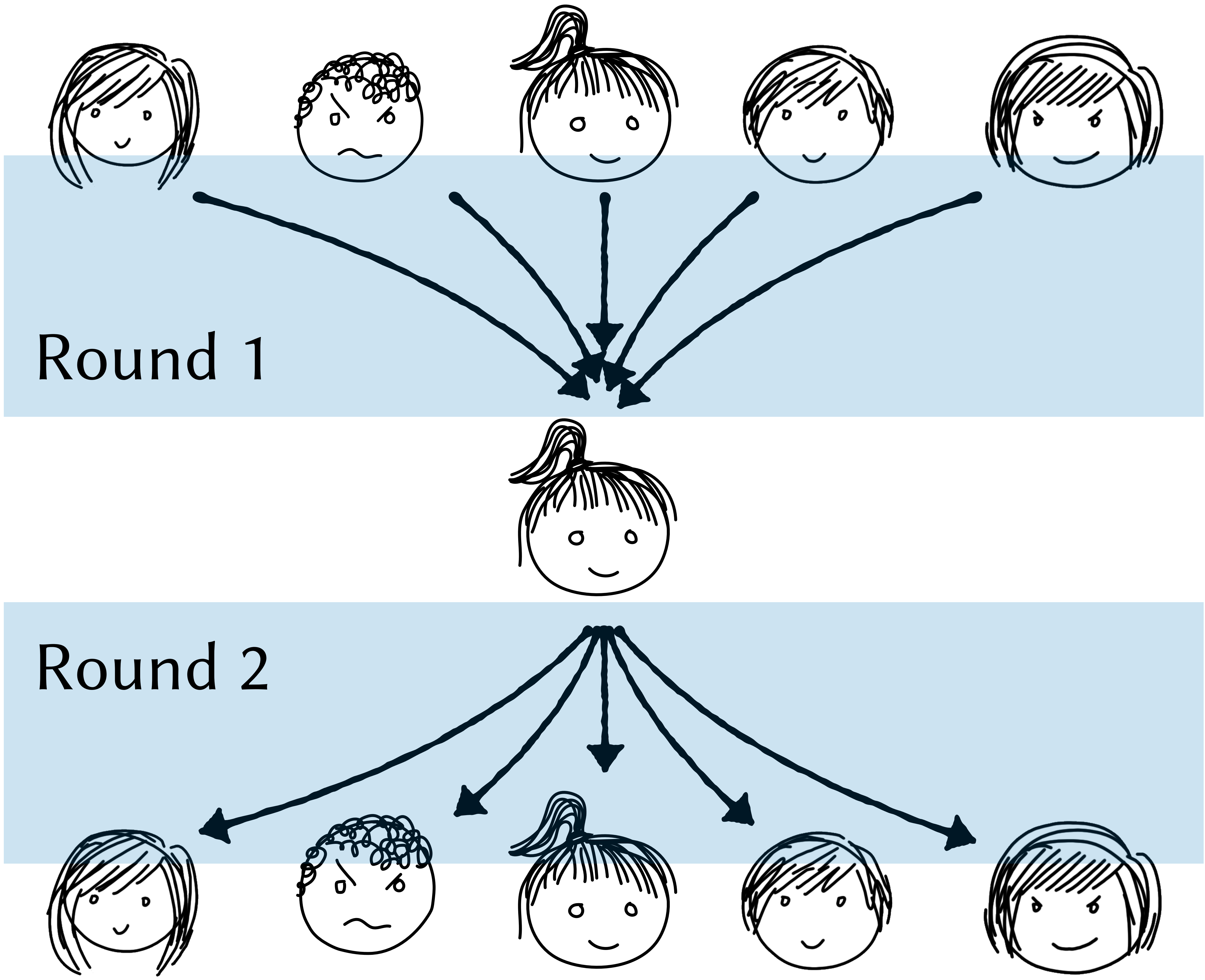


Masking Multiplication

Observation 1: the unwanted terms are linear in w_1, w_2, so maybe we can find a way to remove them without interacting?

Observation 2: Since v_1 is *public*, we can compute $v_1 \cdot \langle r_2 \rangle = \langle w_1 \cdot r_2 - r_1 \cdot r_2 \rangle$ using a non-interactive *scalar* gate. This gives us $v_1 \cdot v_2 + v_1 \cdot \langle r_2 \rangle + v_2 \cdot \langle r_1 \rangle = \langle w_1 \cdot w_2 - r_1 \cdot r_2 \rangle$

Observation 3: if we have to compute (shares of) $r_1 \cdot r_2$ then clearly we are back where we started. The good news is that this term is independent of the secrets. In the *preprocessing* model, we can do it ahead of time!



Preprocessing and Correlated Randomness

- We will enrich our model of protocols slightly by adding an optional *preprocessing* phase to every protocol, which is always run at the start of the experiment. After the this finishes, the parties receive their inputs and the *online* phase begins.
- Usually there is nothing in a functionality that corresponds directly to the preprocessing phase, because the preprocessing phase does not have any IO.
- Typically, in the preprocessing phase, we will talk about receiving *correlated randomness* from a Trusted Dealer. This dealer is an ideal functionality (unless we say e.g. "P_1 is the dealer"). In order to run the protocol in the real world, we need to realize the dealer! *How can we do this generally?*
- *Correlated Randomness* in general means a uniform sample from the set of all values that satisfy some correlation. In this class, a *correlation* is simply a predicate on the values. For example the multiplicative correlation $\mathsf{C}_{\mathsf{mul}}(a, b, c)$ over $\mathbb{F}$ outputs 1 if $a \cdot b = c$, and 0 otherwise. To sample a random element of this correlation means to sample uniformly from $\{(a, b, c) \in \mathbb{F}^3 : \mathsf{C}_{\mathsf{mul}}(a, b, c) = 1\}$.

Preprocessing and Correlated Randomness

- *Correlated Randomness* in general means a uniform sample from the set of all values that satisfy some correlation. In this class, a *correlation* is simply a predicate on the values. For example the multiplicative correlation $\mathsf{C}_{\mathsf{mul}}(a, b, c)$ over $\mathbb{F}$ outputs 1 if $a \cdot b = c$, and 0 otherwise. To sample a random element of this correlation means to sample uniformly from $\{(a, b, c) \in \mathbb{F}^3 : \mathsf{C}_{\mathsf{mul}}(a, b, c) = 1\}$.

- Correlated randomness gives us a nice, general design pattern for protocols: we can often identify a particular correlation that allows us to achieve *information-theoretic security* (which means security against an unbounded adversary - perfect security is the strongest kind of information-theoretic security). We can then reason separately about how to *sample* that correlation, and how to *use* it.

- Typically, *using* correlated randomness is very efficient, which allows us to focus our attention on the best way to sample it.

Putting it Together: $\pi_{\mathsf{Beaver}}(n, t, p)$.

Preprocessing Phase:

1. A trusted dealer samples $(a, b) \leftarrow \mathbb{F}_p^2$ and computes $c := a \cdot b$

2. The then samples $\langle a \rangle \leftarrow \mathsf{Share}_{p,n,t}(a)$, $\langle b \rangle \leftarrow \mathsf{Share}_{p,n,t}(b)$, and $\langle c \rangle \leftarrow \mathsf{Share}_{p,n,t}(c)$, and sends $(\langle a \rangle_i, \langle b \rangle_i, \langle c \rangle_i)$ to every P_i.

   Note: random samples from *this* correlation are called *Beaver Triples*, after Don Beaver, who invented them.

Putting it Together: $\pi_{\mathsf{Beaver}}(n, t, p)$.

Online Phase:

Inputs: Every P_i for $i \in [n]$ has input $\langle w_1 \rangle_i = f_1(i)$ where $f_1 \in \mathscr{P}_{p,t,w}$ and input $\langle w_2 \rangle_i = f_2(i)$ where $f_2 \in \mathscr{P}_{p,t,w_2}$.

1. Every P_i locally computes $\langle v_1 \rangle_i := \langle w_1 \rangle_i - \langle a \rangle_i$ and $\langle v_2 \rangle_i := \langle w_2 \rangle_i - \langle b \rangle_i$ and sends $\langle v_1 \rangle_i$ and $\langle v_2 \rangle_i$ to P_1.

2. P_1 computes $v_1 := \mathsf{Recon}_{p,n,t}([n], \langle v_1 \rangle)$ and $v_2 := \mathsf{Recon}_{p,n,t}([n], \langle v_2 \rangle)$ and sends v_1 and v_2 to all other parties.

3. Every P_i locally computes $\langle w_1 \cdot w_2 \rangle_i := v_1 \cdot v_2 + v_1 \cdot \langle b \rangle_i + v_2 \cdot \langle a \rangle_i + \langle c \rangle_i$.

Outputs: Each P_i ends with $\langle w_1 \cdot w_2 \rangle_i$.

Sketching Security for $\pi_{\mathsf{Beaver}}(n, t, p)$.

Correctness: $\langle w_1 \cdot w_2 \rangle = v_1 \cdot v_2 + v_1 \cdot \langle b \rangle + v_2 \cdot \langle a \rangle + \langle c \rangle$

$$= (w_1 - a) \cdot (w_2 - b) + \langle b \rangle \cdot (w_1 - a) + \langle a \rangle \cdot (w_2 - b) + \langle c \rangle$$

Expand

$$= (w_1 \cdot w_2 - a \cdot w_2 - b \cdot w_1 + a \cdot b) + \langle b \rangle \cdot (w_1 - a) + \langle a \rangle \cdot (w_2 - b) + \langle c \rangle$$

$$= w_1 \cdot w_2 - \langle a \cdot b \rangle + \langle c \rangle$$

Cancel

$$= w_1 \cdot w_2 + \langle 0 \rangle$$

Sketching Security for $\pi_{\mathsf{Beaver}}(n, t, p)$.

Lemma 1: Let $t < n < p$. Assuming synchrony, secure point-to-point channels, and a trusted dealer, $\pi_{\mathsf{Beaver}}(n, t, p)$ perfectly realizes $\mathcal{F}_{\mathsf{mul}}(\mathbb{F}_p)$ in the presence of a semi-honest adversary that statically corrupts up to t parties.

Pf Sketch:

- We can use the simplified security definition for *randomized* functions. In particular, we will show that there exists an algorithm Sim such that for every $I \subseteq [n]$ of size $|I| \leq t$ and every input vector $((x_1, y_1), \ldots, (x_n, y_n))$ such that $(x_1, \ldots, x_n) \in \mathrm{image}(\mathsf{Share}_{p,n,t})$ and $(y_1, \ldots, y_n) \in \mathrm{image}(\mathsf{Share}_{p,n,t})$ we have

$$\left(\mathsf{Sim}\left(I, (\vec{x}_I, \vec{y}_I), \mathsf{mul}_I((x_1, y_1), \ldots, (x_n, y_n))\right), \mathsf{mul}((x_1, y_1), \ldots, (x_n, y_n))\right) \equiv (\mathsf{VIEW}_I, \mathsf{OUTPUT}_{\pi_{\mathsf{Beaver}}})$$

Sketching Security for $\pi_{\mathsf{Beaver}}(n, t, p)$.

- During the preprocessing phase, Sim samples $\langle a \rangle_i \leftarrow \mathbb{F}_p$ and $\langle b \rangle_i \leftarrow \mathbb{F}_p$ for every corrupted P_i. This is perfectly indistinguishable from the real protocol due to the privacy property of Shamir sharing.

- Sim samples $v_1, v_2 \leftarrow \mathbb{F}_p$, computes $\langle v_1 \rangle_i := \langle w_1 \rangle_i - \langle a \rangle_i$ and $\langle v_2 \rangle_i := \langle w_2 \rangle_i - \langle b \rangle_i$ for every corrupt P_i, and then samples $\langle v_1 \rangle_h$ and $\langle v_2 \rangle_h$ for $h \in [n] \backslash I$ uniformly subject to the constraint that $\langle v_1 \rangle$ and $\langle v_2 \rangle$ are valid degree-t Shamir sharings of v_1 and v_2 respectively.

- Sim adds $\langle v_1 \rangle$ and $\langle v_2 \rangle$ to the view of P_1 if it is corrupted; v_1, v_2 are added to the view of every corrupted P_i. This is perfectly indistinguishable from the protocol because $a, b \leftarrow \mathbb{F}_p$ in the protocol, which implies that v_1, v_2 are uniform, and the other values adhere to the same set of constraints in both the real and ideal worlds.

Sketching Security for $\pi_{\mathsf{Beaver}}(n, t, p)$.

- Sim adds $\langle v_1 \rangle$ and $\langle v_2 \rangle$ to the view of P_1 if it is corrupted; v_1, v_2 are added to the view of every corrupted P_i. This is perfectly indistinguishable from the protocol because $a, b \leftarrow \mathbb{F}_p$ in the protocol, which implies that v_1, v_2 are uniform, and the other values adhere to the same set of constraints in both the real and ideal worlds.

- The above values together with the output $\langle w_1 \cdot w_2 \rangle_i$ of each corrupted party (which is an input to Sim) fix a particular value of $\langle c \rangle_i$ for each corrupted P_i, per the equation $\langle w_1 \cdot w_2 \rangle_i = v_1 \cdot v_2 + v_1 \cdot \langle b \rangle_i + v_2 \cdot \langle a \rangle_i + \langle c \rangle_i$. Because $\langle w_1 \cdot w_2 \rangle_i$ is distributed uniformly in the ideal world (mul samples it using the $\mathsf{Share}_{p,n,t}$ algorithm), $\langle c \rangle_i$ is too. In the real world, is $\langle c \rangle_i$ sampled by the dealer using the $\mathsf{Share}_{p,n,t}$ algorithm, and $\langle w_1 \cdot w_2 \rangle_i$ is computed via the same equation. Thus the distributions are identical.

Sketching Security for $\pi_{\mathsf{Beaver}}(n, t, p)$.

- The above values together with the output $\langle w_1 \cdot w_2 \rangle_i$ of each corrupted party (which is an input to Sim) fix a particular value of $\langle c \rangle_i$ for each corrupted P_i, per the equation $\langle w_1 \cdot w_2 \rangle_i = v_1 \cdot v_2 + v_1 \cdot \langle b \rangle_i + v_2 \cdot \langle a \rangle_i + \langle c \rangle_i$. Because $\langle w_1 \cdot w_2 \rangle_i$ is distributed uniformly in the ideal world (mul samples it using the $\mathsf{Share}_{p,n,t}$ algorithm), $\langle c \rangle_i$ is too. In the real world, is $\langle c \rangle_i$ sampled by the dealer using the $\mathsf{Share}_{p,n,t}$ algorithm, and $\langle w_1 \cdot w_2 \rangle_i$ is computed via the same equation. Thus the distributions are identical.

- $\langle c \rangle_i$ is inserted into the view of P_i during the preprocessing phase. Remember that Sim is not required to generate the view in the same order as the real protocol does!

- Output consistency is guaranteed because Sim programs $\langle w_1 \cdot w_2 \rangle_i$ for every corrupt P_i to be precisely the value that it is given as input, and by the fact that the protocol is correct. This completes the theorem. ∎

Why did we do this?

Corollary 1: Let $t < n < p$. Assuming synchrony, secure point-to-point channels, and a trusted dealer, there exists an n-party protocol in the preprocessing model that perfectly realizes $\mathcal{F}_{\mathsf{SFE}}$ in the presence of a semi-honest adversary that statically corrupts up to t parties. The online bandwidth complexity of this protocol is in $O((c_{\mathsf{in}} + c_{\mathsf{out}} + c_{\times}) \cdot n \cdot |p|)$.

Notice: Because we used a trusted dealer we were able to prove security against a *dishonest majority* (i.e. $t < n$), whereas with the BGW and GRR multiplication protocols we required an *honest majority* ($2t < n$).

Unfortunately, we will find that if we try to perfectly realize the dealer, we will require an honest majority to do so. The requirement for an honest majority turns out to be *inherent* in perfectly-secure protocols that compute nonlinear functions!

(Chalkboard Proof)

So, where do we go from here?

To achieve security against a dishonest majority, we must give up something.

Another Look at the Setting.

Perfect MPC Feasibility Theorem: Let $2t < n < p$. Assuming synchrony and secure point-to-point channels there exists an n-party protocol that perfectly realizes $\mathcal{F}_{\mathsf{SFE}}$ in the presence of a semi-honest adversary that statically corrupts up to t parties.

- Static semi-honest adversaries already have the weakest corruption behavior. In fact, we would like very much to handle *stronger* adversaries than this.
- A fully connected, synchronous network of secure point-to-point channels is already the strongest network assumption we could make.
- The only other piece of this statement that we can change is the power of the adversary. If we want to handle $n/2 \leq t < n$, then we *must* give up perfect security and settle for something weaker.
- Before we talk about exactly what kind of security we can achieve and how we must change our model, I want to show you what kind of functionality we must realize. In other words, what is the shape of the missing puzzle piece that we need to find?

Let's Start From the BGW Protocol

- The foundation of this protocol was Shamir's $(t+1)$-out-of-n secret sharing scheme.
- Ultimately, in order to obtain correct multiplication protocols, we had to set $t < n/2$, and today we proved that this wasn't some quirk of the particular protocols we explicitly constructed: this bound is actually *necessary* in order to achieve perfect security.
- However, recall that the BGW protocol was secure against even $t = n - 1$ corruptions in the $\mathcal{F}_{\mathsf{mul}}$-hybrid model. Our problem was really with multiplication, the other parts of the protocol seem to be OK.
- If we're going to set $t \geq n/2$, we might as well go all the way to $t = n - 1$. This means that our secret sharing scheme becomes an n-out-of-n scheme, and we can use *additive secret sharing* instead of Shamir sharing (no more polynomials!).

Simplifying the Problem

- Switching to additive sharing means that we no longer need to work over some $\mathbb{F}_p$ such that $p > n$. So for the time being, let's think about $\mathbb{F}_2$. We will return to the general case later. We can also set $n = 2$, which was previously impossible.
- This setting is simple and essential: two parties want to compute a boolean function together. We would probably start our class with this scenario if it weren't for the fact that we can prove that it requires more advanced techniques.
- What does our protocol look like now? All wire values and shares are single bits.
  - To **input** x_i from P_i on wire o, P_i samples $\langle w_o \rangle \leftarrow \mathsf{Share}_{2,2,2}(x_i)$ and sends $\langle w_o \rangle_{1-i}$ to P_{1-i}. Notice that $\langle w_o \rangle_1 \oplus \langle w_o \rangle_2 = x_i$.
  - To set wire o's value as the **sum** of wires j and k, P_i for $i \in [2]$ computes $\langle w_o \rangle_i := \langle w_j \rangle_i \oplus \langle w_k \rangle_i$ non-interactively.
  - To **output** wire j to P_i, P_{1-i} sends $\langle w_o \rangle_{1-i}$ to P_i, who computes $y_i := \langle w_o \rangle_i \oplus \langle w_o \rangle_{1-i}$.

Simplifying the Problem

- What does our protocol look like now? All wire values and shares are single bits.
  - To set wire o's value as the **product** of wires j and k, P_i for $i \in [2]$ sends $\langle w_j \rangle_i$ and $\langle w_k \rangle_i$ to $\mathcal{F}_{\mathsf{mul}}$.
  - $\mathcal{F}_{\mathsf{mul}}$ reconstructs w_j and w_k, and computes $w_o := w_j \wedge w_k$. Then it samples $r \leftarrow \{0,1\}$ and sends $\langle w_o \rangle_1 := r$ to P_1 and $\langle w_o \rangle_2 := r \oplus w_o$ to P_2.
  - Suppose that $\mathcal{F}_{\mathsf{mul}}$ samples two values, $r_{12}, r_{21} \leftarrow \{0,1\}$ and sets $r := r_{12} \oplus r_{21}$...
  - We can then inline the reconstruction and multiplication equations to find that $\mathcal{F}_{\mathsf{mul}}$ computes $\langle w_o \rangle_2 := r_{12} \oplus r_{21} \oplus \left(\langle w_j \rangle_1 \oplus \langle w_j \rangle_2 \right) \wedge \left(\langle w_k \rangle_1 \oplus \langle w_k \rangle_2 \right)$

Simplifying the Problem

$$\langle w_o \rangle_2 := r_{12} \oplus r_{21} \oplus \left(\langle w_j \rangle_1 \oplus \langle w_j \rangle_2\right) \wedge \left(\langle w_k \rangle_1 \oplus \langle w_k \rangle_2\right)$$

$$= \langle w_j \rangle_1 \wedge \langle w_k \rangle_1 \oplus r_{12} \oplus \langle w_j \rangle_1 \wedge \langle w_k \rangle_2 \oplus r_{21} \oplus \langle w_j \rangle_2 \wedge \langle w_k \rangle_1 \oplus \langle w_j \rangle_2 \wedge \langle w_k \rangle_2$$

P_1 Can compute this locally

These terms require interaction

P_2 Can compute this locally

Since we masked them individually, we can compute them separately.

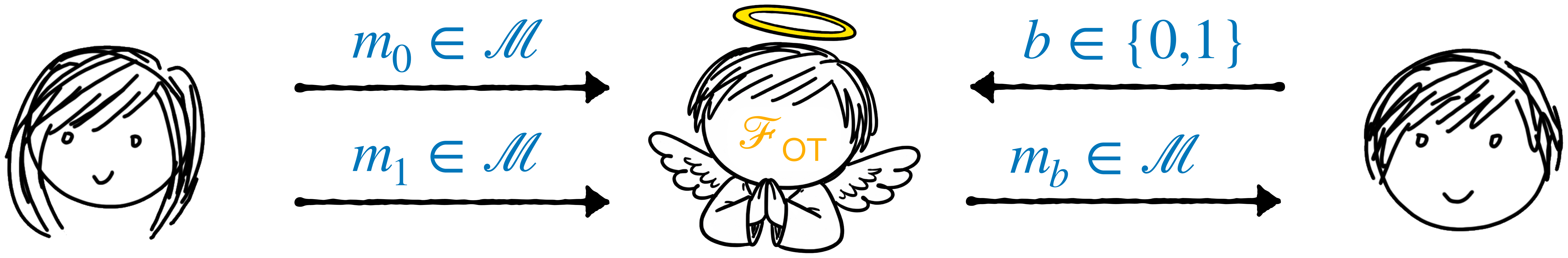
Notice that they are the AND of one bit that P_1 knows and one bit that P_2 knows, and they are one-time-padded with a bit that P_1 knows.

A New Functionality?

- Let's think about this task abstractly.
- Suppose we want to take an input $a \in \{0,1\}$ from P_1 and an input $b \in \{0,1\}$ from P_2, and then output a random $r \leftarrow \{0,1\}$ to P_1, and $v := r \oplus a \wedge b$ to P_2.
- This is precisely what we proved we could not perfectly securely compute, earlier.
- P_1 only learns a random value. *What does P_2 learn?*
- P_2's input gives it a choice! If $b = 0$, then P_2 learns r. If $b = 1$, then P_2 learns $r \oplus a$. If r is uniform, then these values appear identically distributed to P_2.
- Forget for a moment how this task integrates into a bigger protocol. Suppose we ask P_1 to sample r and provide it as a second input, instead of guaranteeing the uniformity of r using an ideal process. *What can go wrong? Can P_1 choose a distribution for r that lets it learn something additional about P_2's inputs or outputs?*
- By choosing a bad r, P_1 can only hurt itself (at least when we ignore larger context), so it seems safe to let P_1 choose r freely.

Introducing *Oblivious Transfer.*

- Let $\mathscr{M}$ be some message space, and $m_0, m_1 \in \mathscr{M}$ be two messages known to P_1.
- The *Oblivious Transfer* functionality allows P_2 to receive *one* message of its choice.
- P_1 isn't allowed to learn which of the two messages P_2 received.
- P_2 isn't allowed to learn anything about the message that it didn't choose.



CS4501 Cryptographic Protocols
Lecture 11: 🦫🦫🦫, The Limits of Perfect MPC

https://jackdoerner.net/teaching/#2026/Spring/CS4501