Garbled Computation:

Hiding Software, Data & Computed Values using Light-Weight Primitives.

Shoaib Khan
Mikhail Atallah
Qutaibah Malluhi

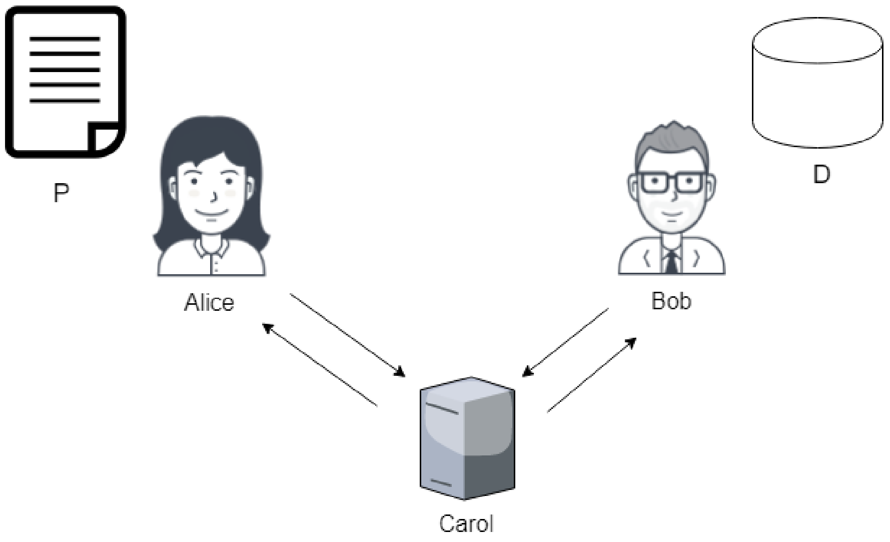
Outline

- Problem Statement
- Background
- The Computational Model (OISC)
- Definitions & Notation
- An Overview of Our Protocol
- Primitives
  - Oblivious Shuffle
  - Oblivious Translation
- Details of Our Protocol (EST)
  - Execute
  - Shuffle
  - Translate
- Implementation
- Q & A

- **Scenario**: Alice has proprietary software P that it wants to run (securely) on confidential data D owned by Bob, possibly using some external computing resources (e.g. Cloud Services).

- **Security:** Nothing about either party's input is revealed to any of the other participants during program execution. All computed values should remain hidden. [In a non-colluding, passive adversary model]

- **Correctness:** At the end of execution, results are correctly computed and revealed only to the desired party.



Problem Statement

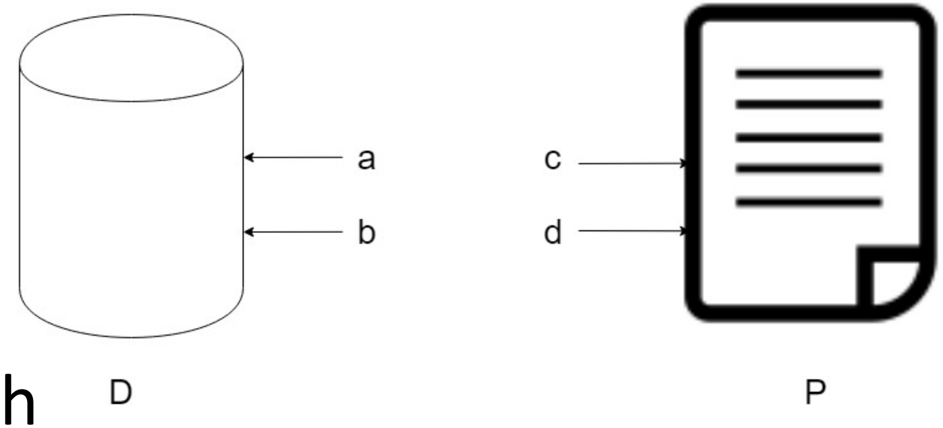
Background

- Computing a publicly known function on confidential data:
  - Yao’s Garbled Circuit
  - Secure Multiparty Computation
  - Homomorphic Encryption

- Computing a confidential function on confidential data:
  - Universal Turing Machine model: Encrypt both function and data and run simulation on UTM
    - Practically infeasible
  - Blind & Permute with Paillier Encryption, Garbled Circuit Evaluation & Oblivious RAM.
    - Involves modular exponentiation & ORAM overhead
    - Does not scale well to large problem sizes.

- In our protocol, we develop Light-Weight Crypto Primitives with the goal of making it practically feasible & scalable, avoiding above mentioned computational overheads.

The Computational Model (OISC)

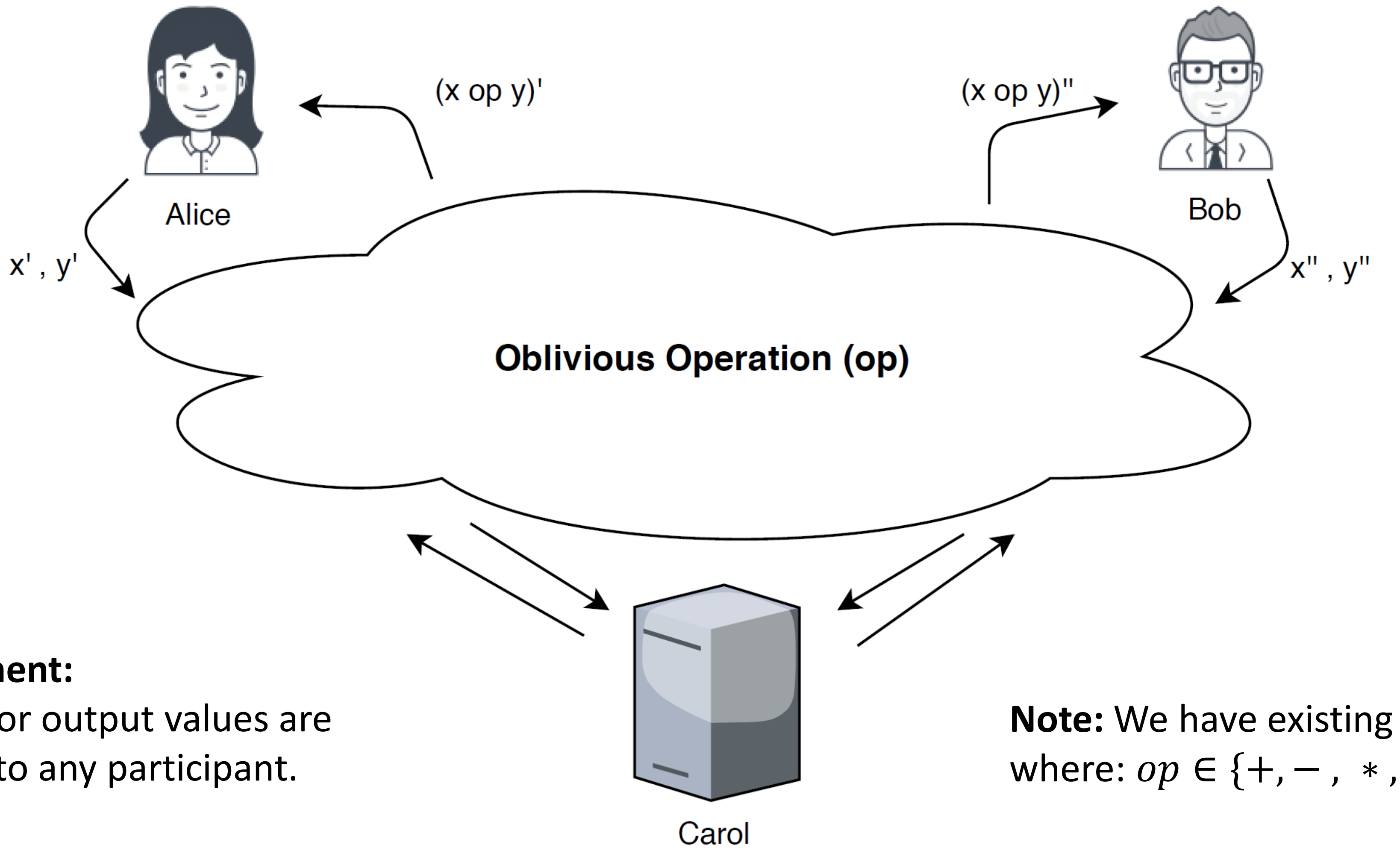


- We use **One Instruction Set Computer** as our model of computation. Why?
- A program P is a sequence of n Turing-Complete (SUBBLE) instructions, with an associated data array D of size m.
- A SUBBLE instruction is a quadruple (a, b, c, d), where $0 \leq a, b < m$ are data addresses and $0 \leq c, d \leq n$ are instruction addresses. (Note: The last instruction in P is $NULL$)
- To execute one instruction: first $D[a] \coloneqq D[a] - D[b]$; then if $D[a] \leq 0$, the next instruction is $P[c]$ else it is $P[d]$.

Definitions & Notation

- *Additive Split of x:* Split x into two random-looking values x' & x'' s.t. $x' + x'' = x$.
- A*dditive split* of a vector $(v_1, \ldots, v_k) = (v_1', \ldots, v_k{}') + (v_1'', \ldots, v_k{}'')$ where for all $i \in [k]$, $v_i = v_i' + v_i{}''$ s.t $v_i', v_i{}''$ are randoms.
- Notation:
  - Alice's Additive Share of any object x is denoted by x'.
  - Bob's Additive Share of any object x is denoted by x''.

Definitions & Notation



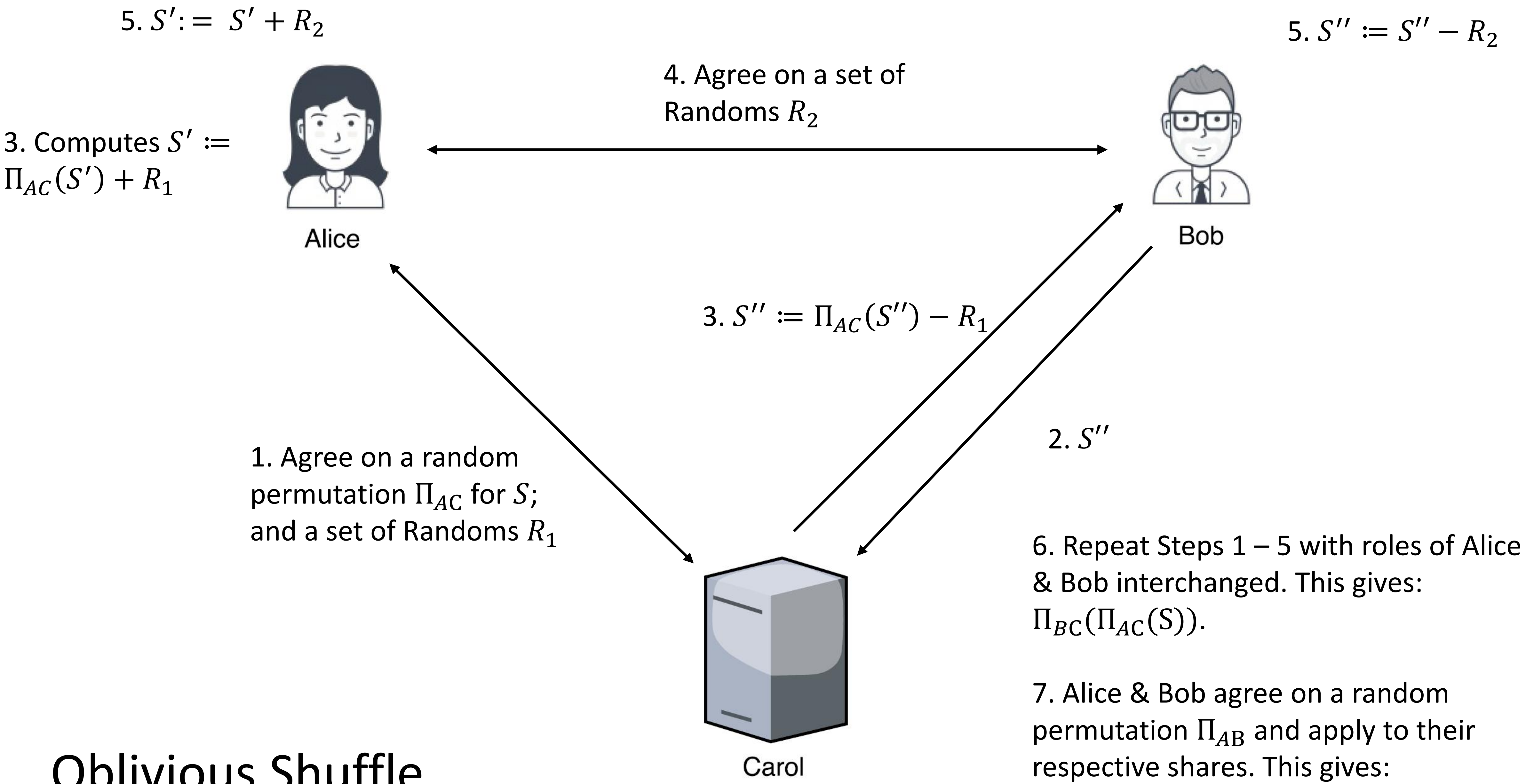
Requirement:
No input or output values are revealed to any participant.

Note: We have existing primitives where: $op \in \{+, -, *, \geq, >, ==\}$

Our Protocol : An Overview

- Three Parties: Alice, Bob and Carol.
- Round-based, where each round involves three steps: i) ***Execute*** (one instruction), ii) ***Shuffle*** (P and D), and iii) ***Translate***.
- We use **Oblivious Comparison($\geq$)** and **Oblivious Subtraction** primitives to execute a SUBBLE instruction in a split manner.
- Followed by **Oblivious Shuffle** which Permutes and Re-splits P and D.
- Followed by **Oblivious Translation** which updates instruction and data addresses in the shuffled program $\pi(P)$.
- **Note:** The number of rounds is proportional to the time complexity of program P on data D if it were executed on a conventional computer.

Oblivious Shuffle of a Sequence S



Need for Translation

- After Oblivious Shuffle, all the addresses in the Program are wrong because they pertain to the pre-shuffle situation.

- Therefore, there is a need for a step that updates all the program addresses to reflect the shuffling that has taken place.

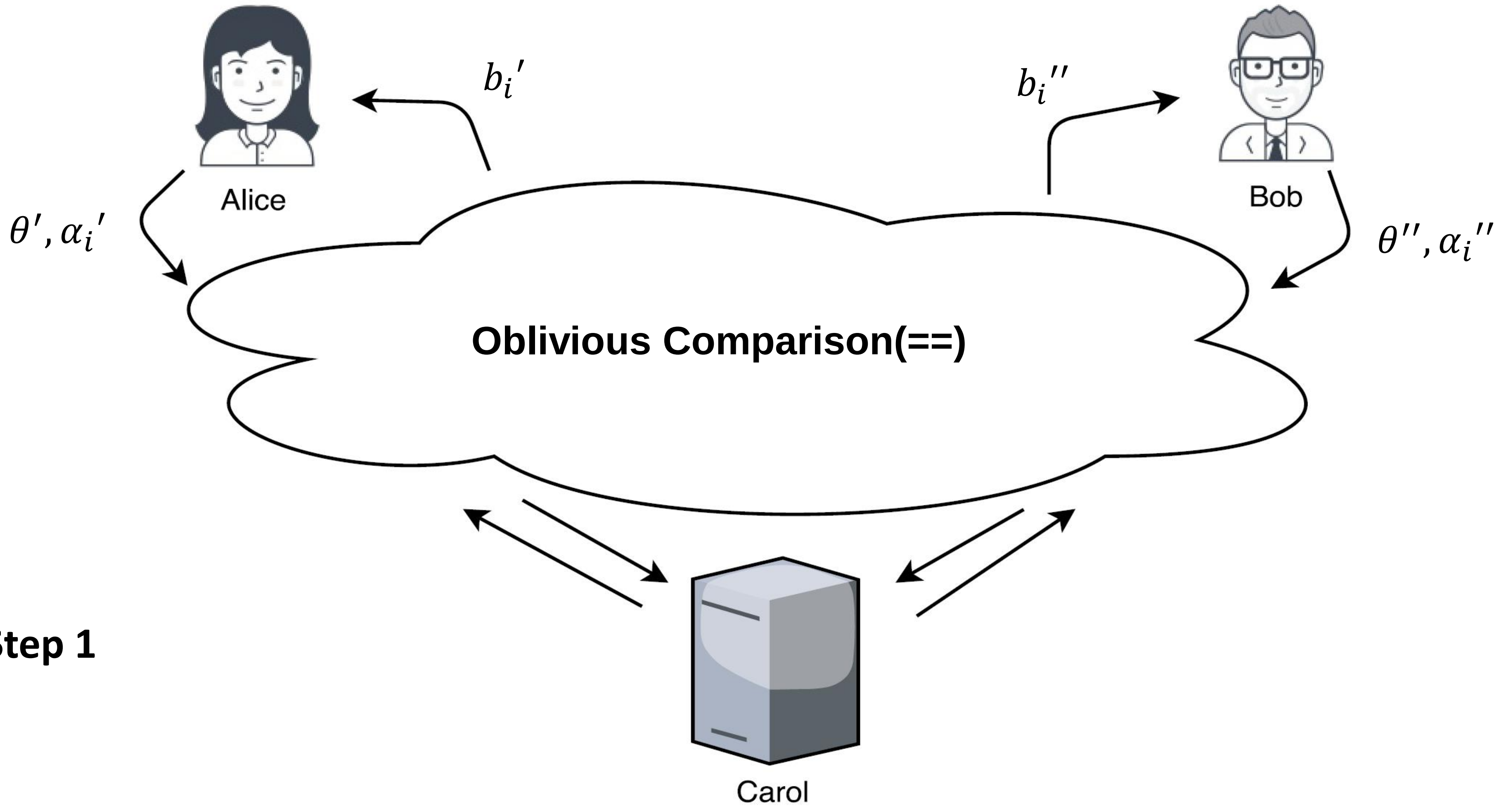
- It is this update of program addresses that is carried out by what we call the Oblivious Translate Protocol.

Oblivious Translation

- **Setup:** Two parties: Alice & Bob.
- The following is done in parallel for all addresses to be translated.
- **Inputs:**
  - All values are additively randomly split between Alice & Bob.
  - A query value θ, that represents a pre-shuffle address to be translated.
  - n pairs of numbers $\langle \alpha_i , \beta_i \rangle$ that define the shuffle's permutation π.
- Note: $\pi(\alpha_i) = \beta_i$ where neither A nor B know π.
- A & B want to compute $\pi(\theta)$, without revealing their shares to anyone, & without learning π.
- **Output:** Alice gets her share $\pi(\theta)'$ and Bob gets his share $\pi(\theta)''$ of the translated query value $\pi(\theta)$.
- Alice & Bob will need Carol's help to complete the protocol.

Intuition: Alice and Bob, with Carol's help, obliviously compute: $\Sigma_{\mathrm{i=1}}^{\mathrm{n}}(\theta == \alpha_i)\beta_{\mathrm{i}}$. Note that this value is equal to $\pi(\theta)$.

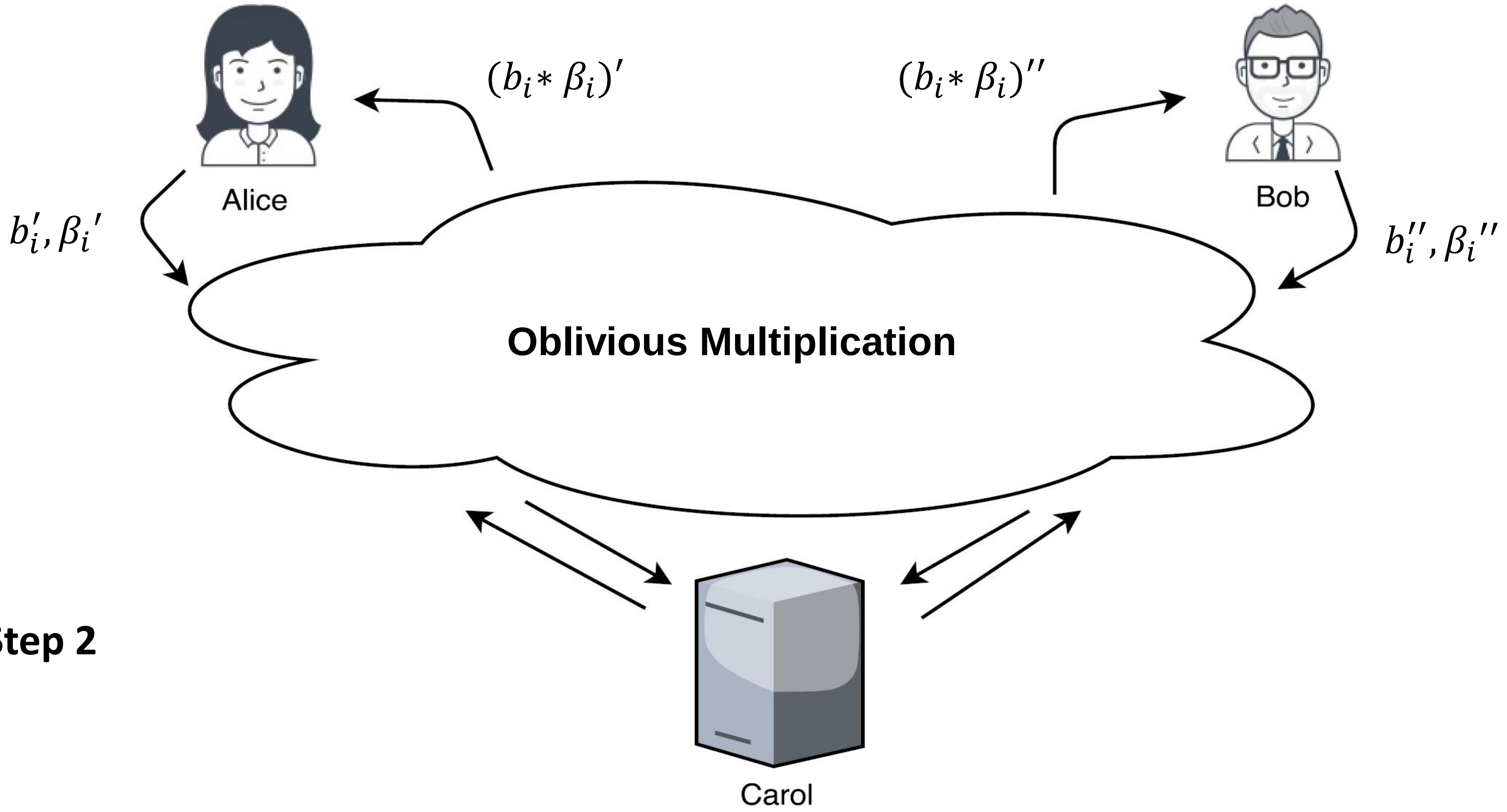
Protocol: For each i from 1 to n :



Step 1

Intuition: Alice and Bob, with Carol's help, obliviously compute: $\Sigma_{\mathrm{i}=1}^{\mathrm{n}}(\theta == \alpha_i)\beta_{\mathrm{i}}$. Note that this value is equal to $\pi(\theta)$.

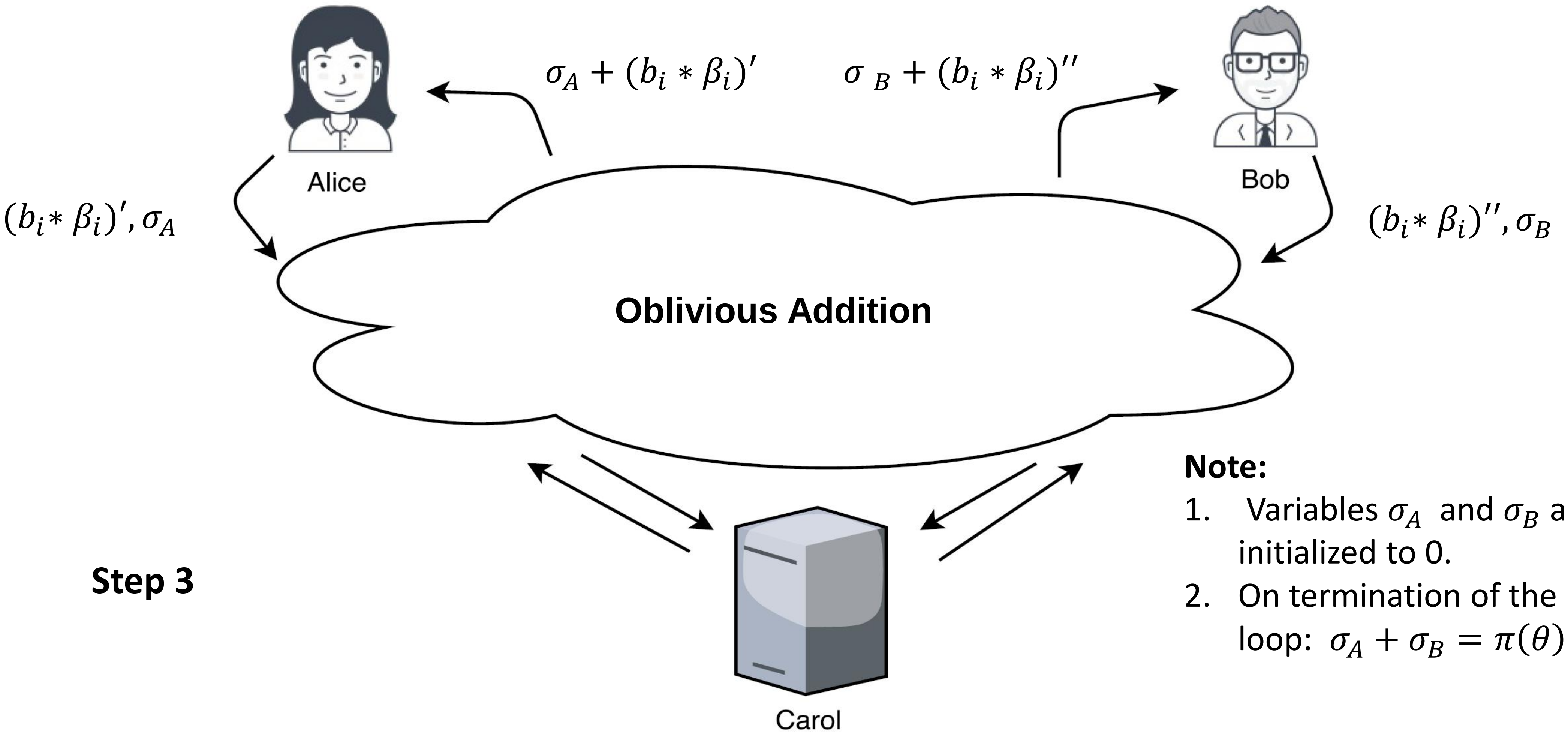
Protocol: For each i from 1 to n :



Step 2

Intuition: Alice and Bob, with Carol's help, obliviously compute: $\Sigma_{\mathrm{i=1}}^{\mathrm{n}}(\theta == \alpha_i)\beta_{\mathrm{i}}$. Note that this value is equal to $\pi(\theta)$.

Protocol: For each i from 1 to n :

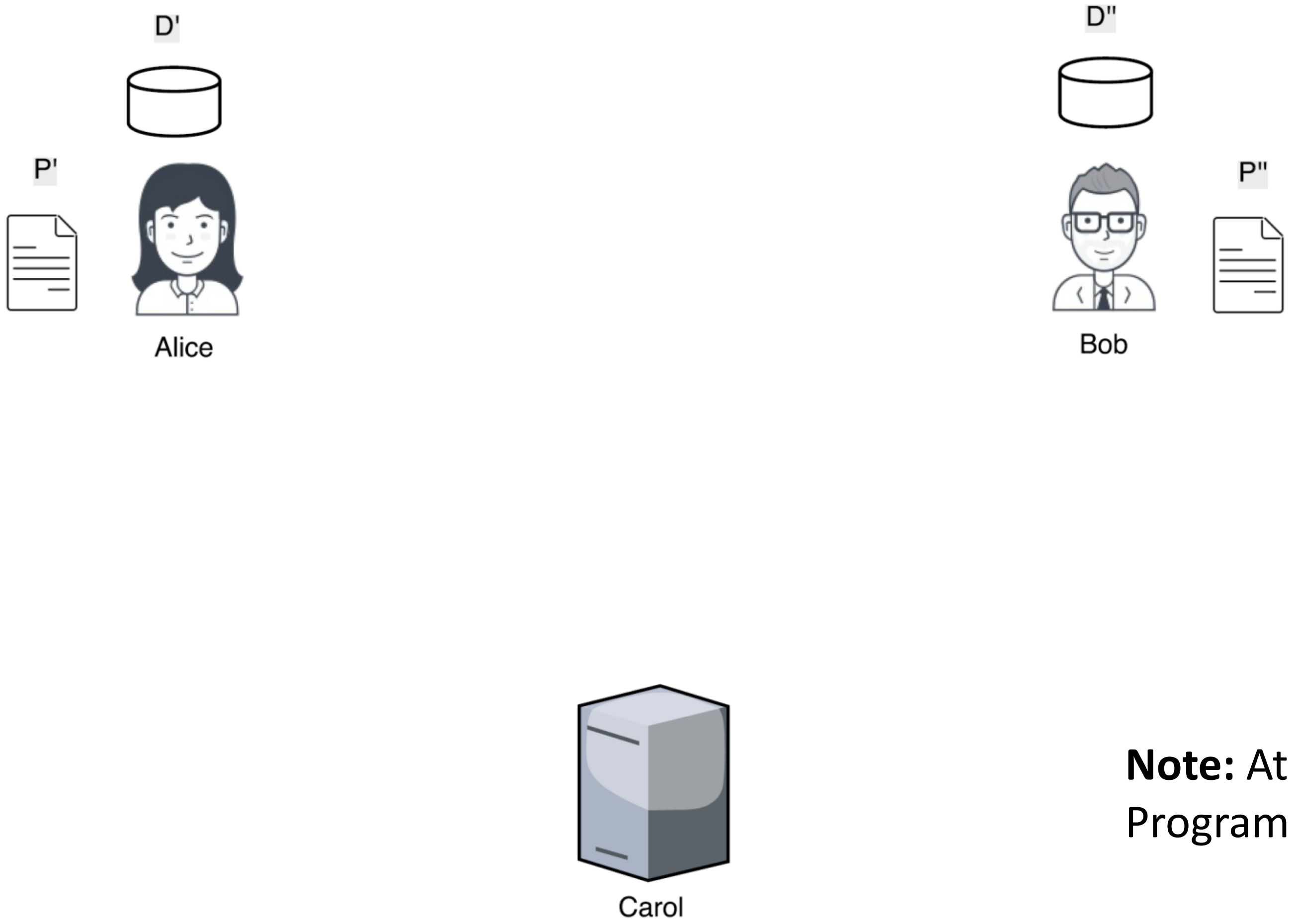


Step 3

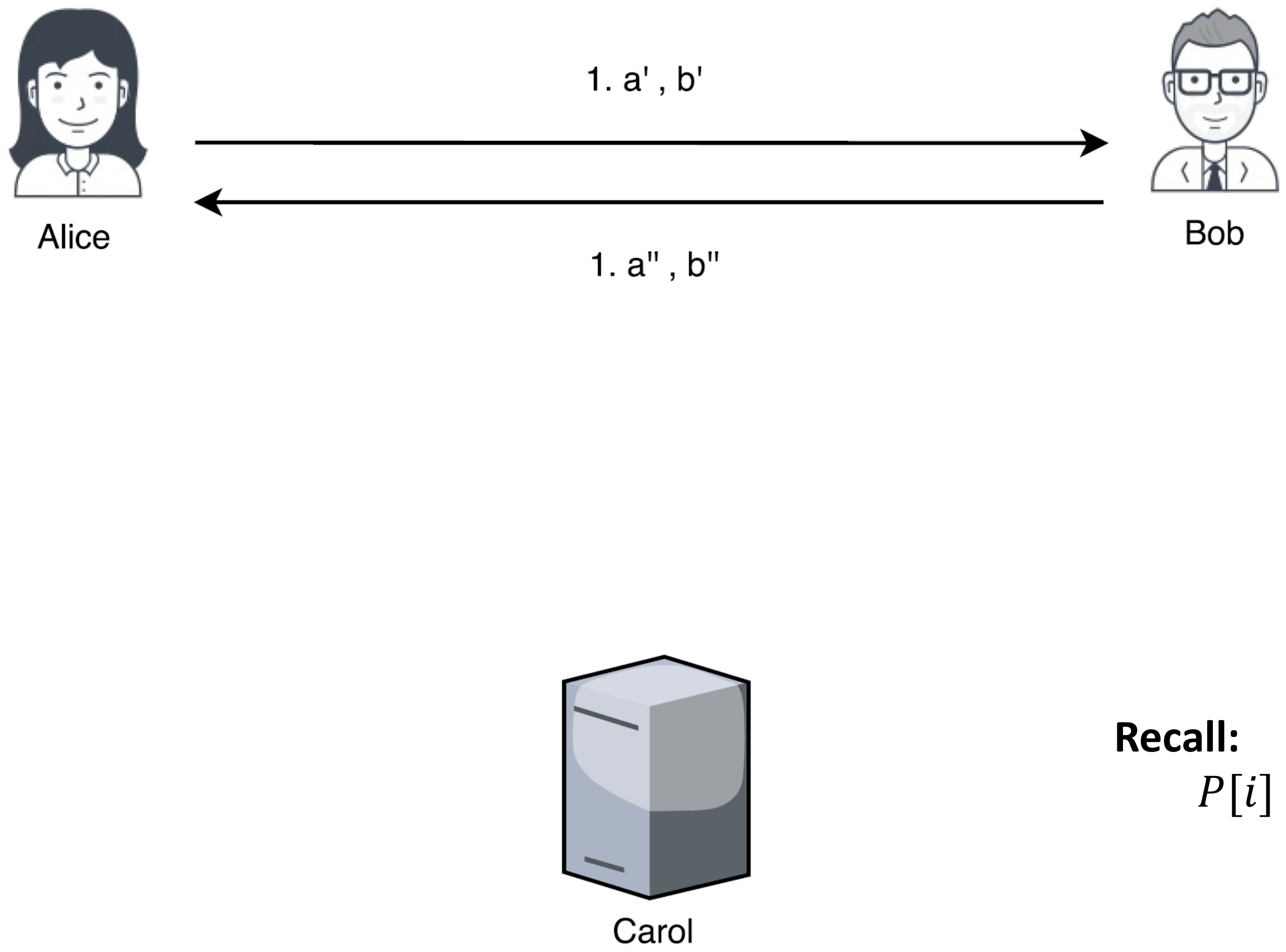
Note:

1. Variables σ_A and σ_B are initialized to 0.
2. On termination of the loop: $\sigma_A + \sigma_B = \pi(\theta)$

Putting the Pieces Together: Complete GC Protocol

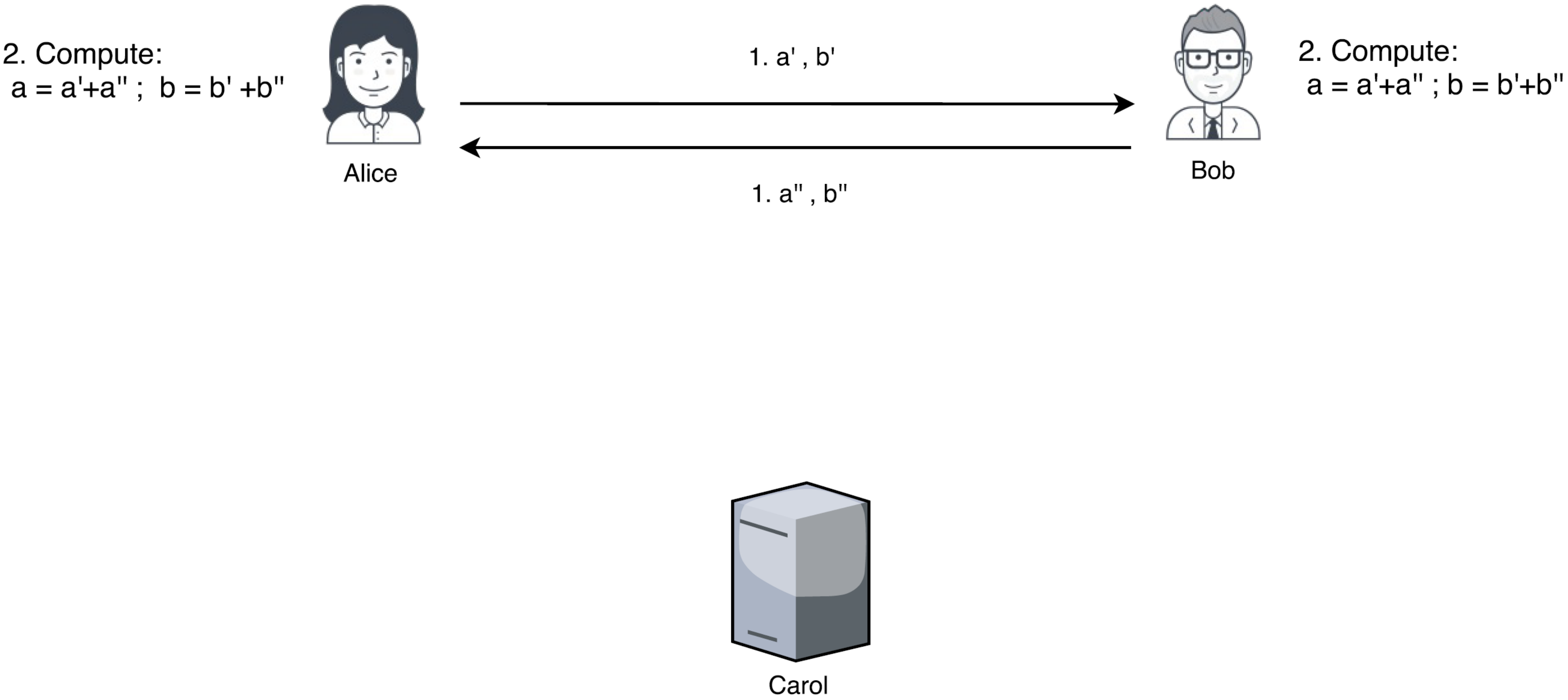


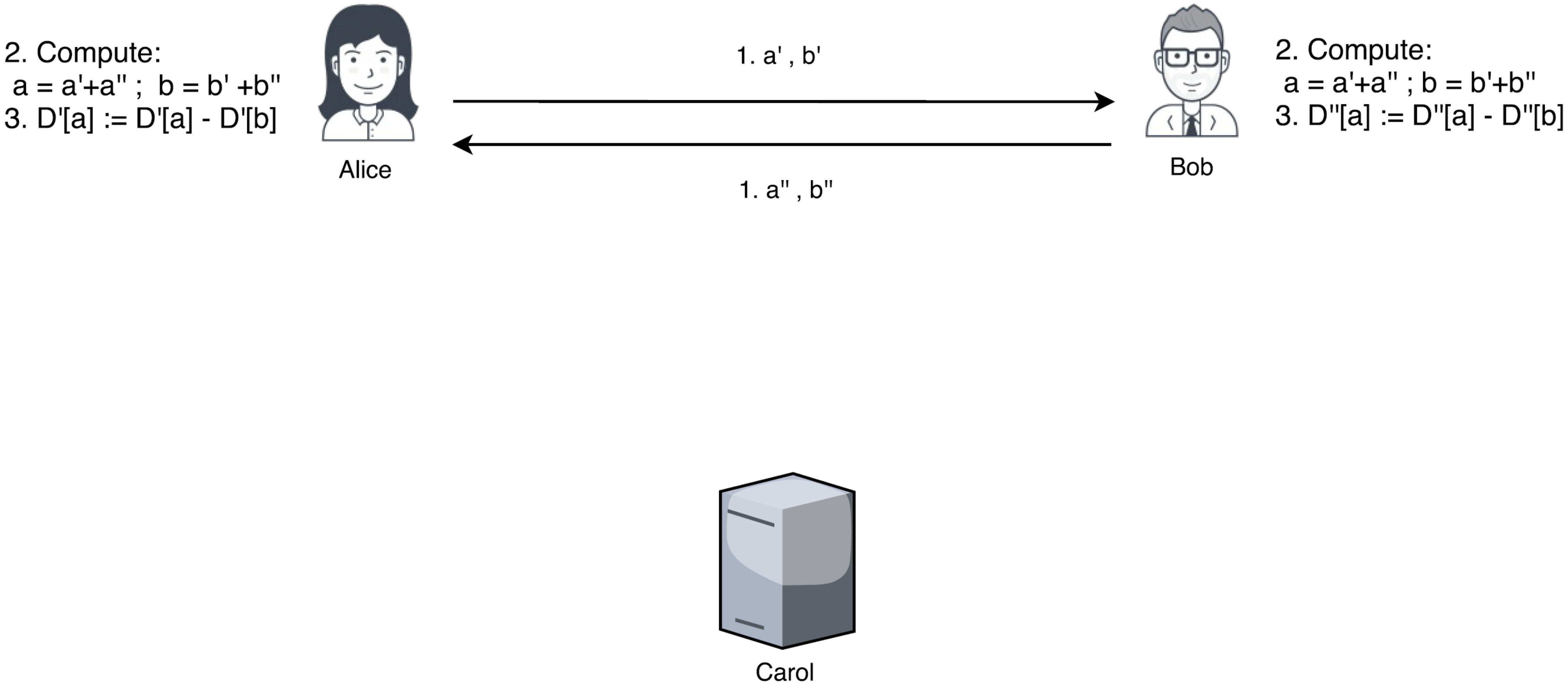
Note: At the start of protocol
Program Counter: $i = 0$.

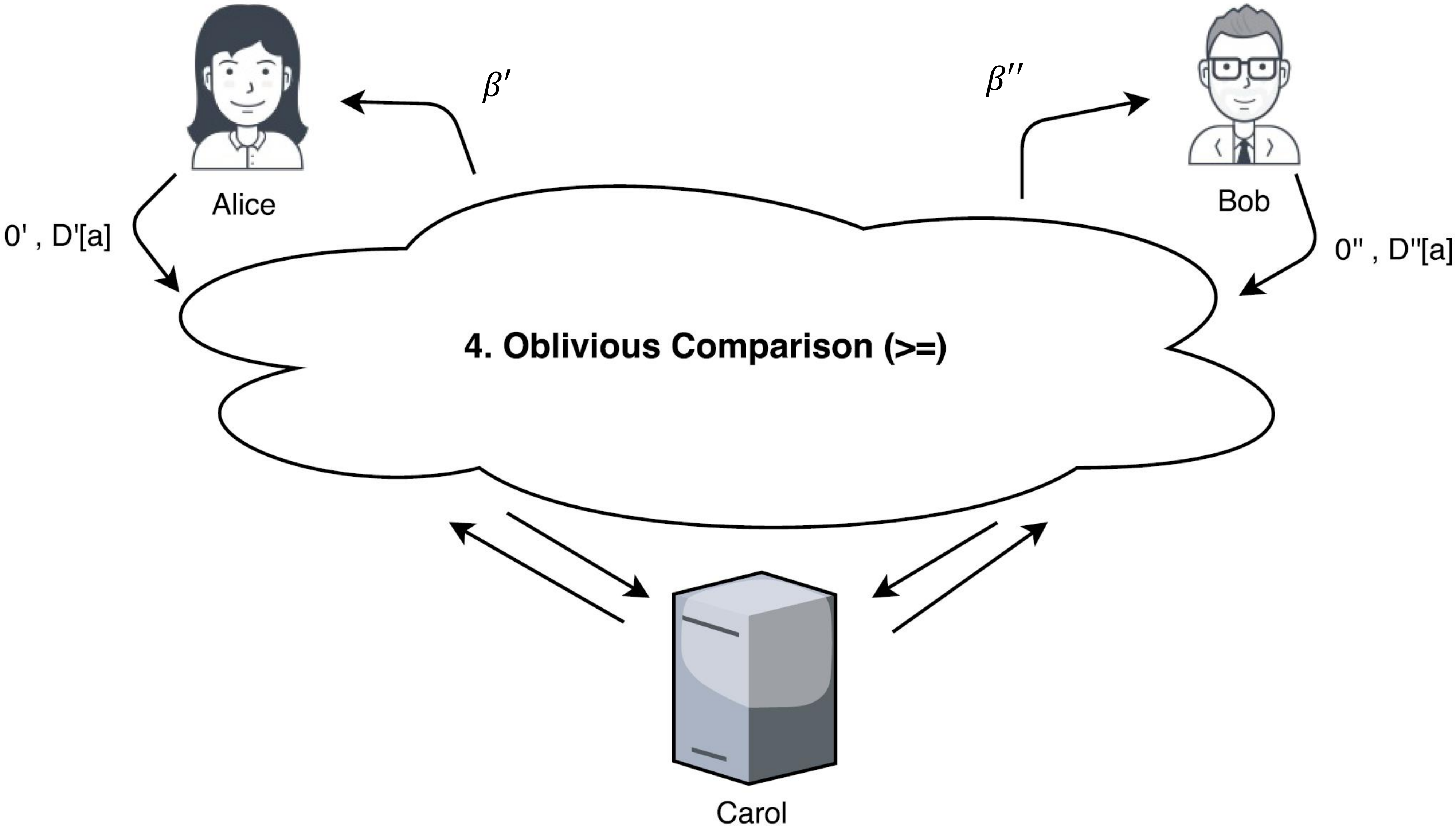


Recall:

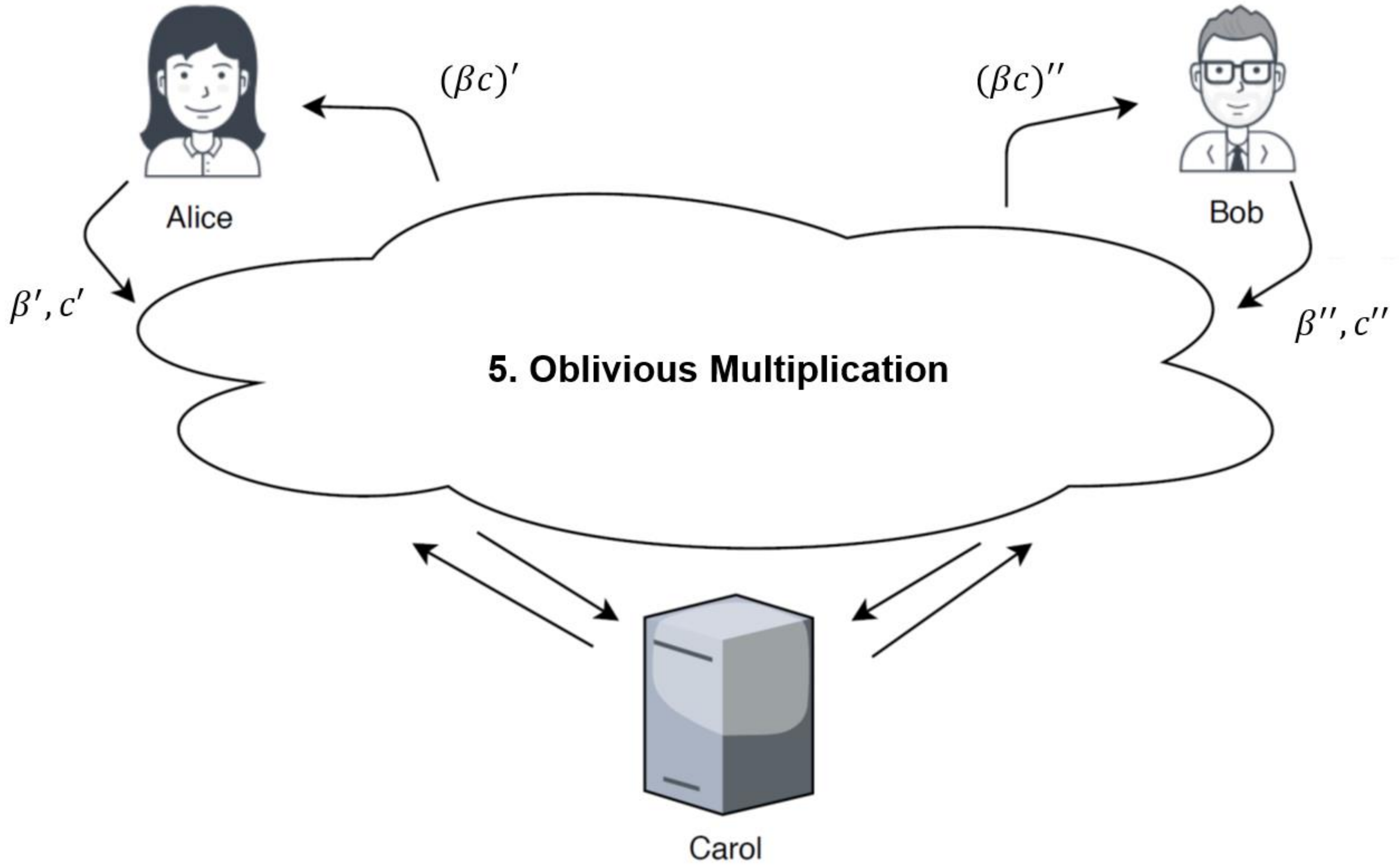
$$P[i] \ = (a, b, c, d)_i$$


2. Compute:
a = a'+a'' ; b = b' +b''
1. a' , b'
2. Compute:
a = a'+a'' ; b = b'+b''
Alice
Bob
1. a'' , b''
Carol

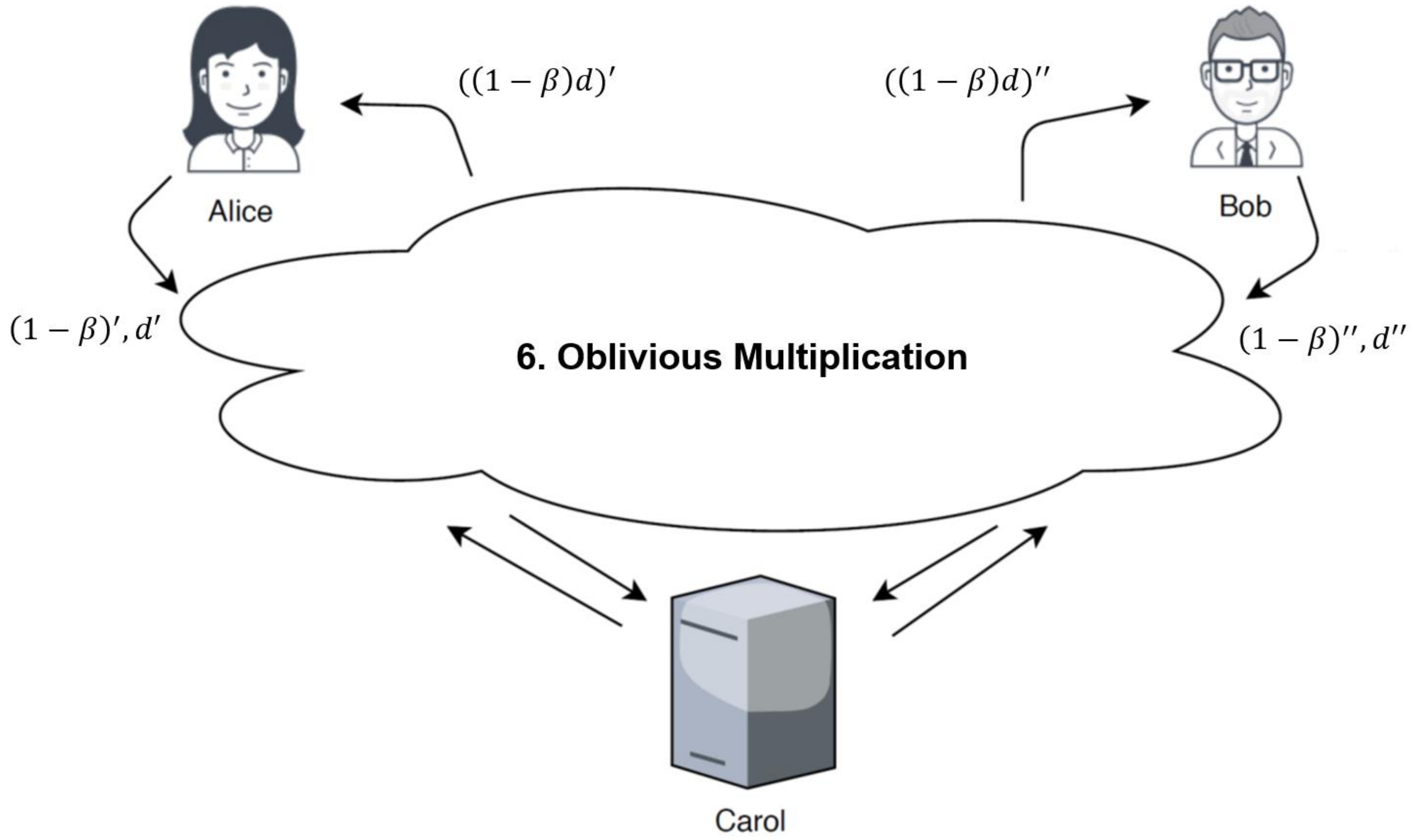

2. Compute:
a = a'+a'' ; b = b' +b''
3. D'[a] := D'[a] - D'[b]
Alice
1. a' , b'
1. a'' , b''
Bob
2. Compute:
a = a'+a'' ; b = b'+b''
3. D''[a] := D''[a] - D''[b]
Carol



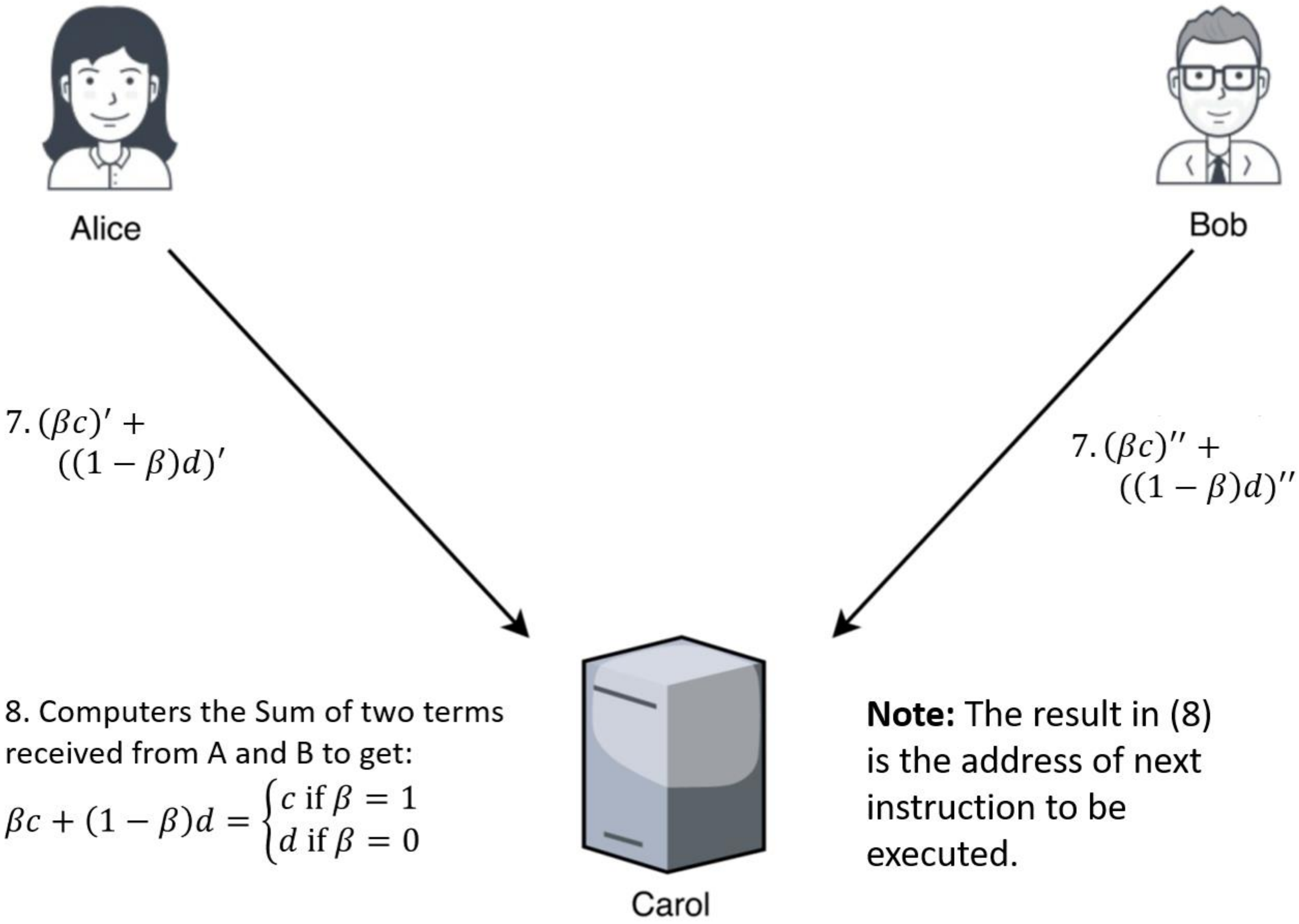
Alice
Bob
β'
β''
0' , D'[a]
0'' , D''[a]
4. Oblivious Comparison (>=)
Carol

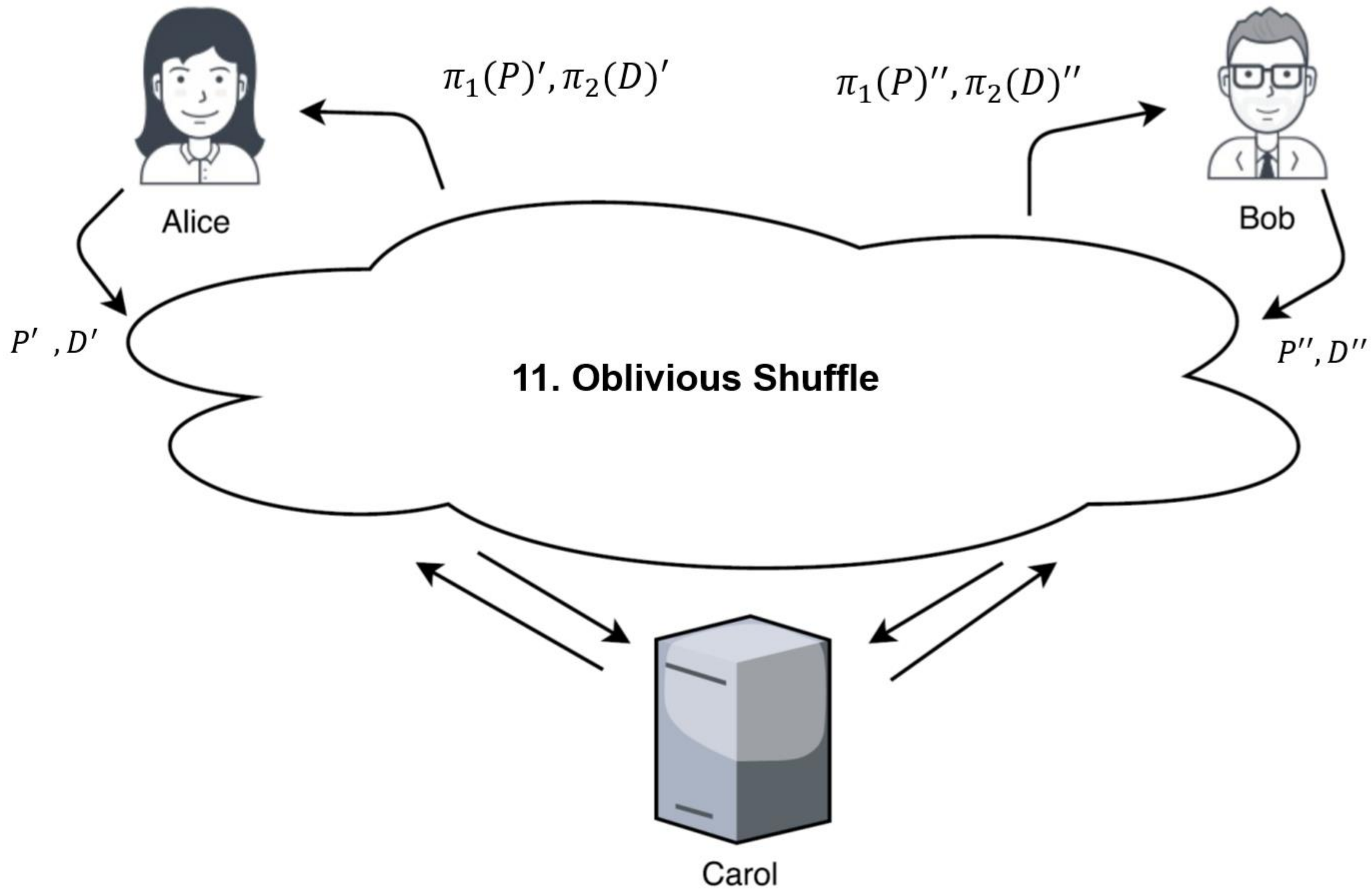


Alice
Bob
$(\beta c)'$
$(\beta c)''$
β', c'
β'', c''
5. Oblivious Multiplication
Carol

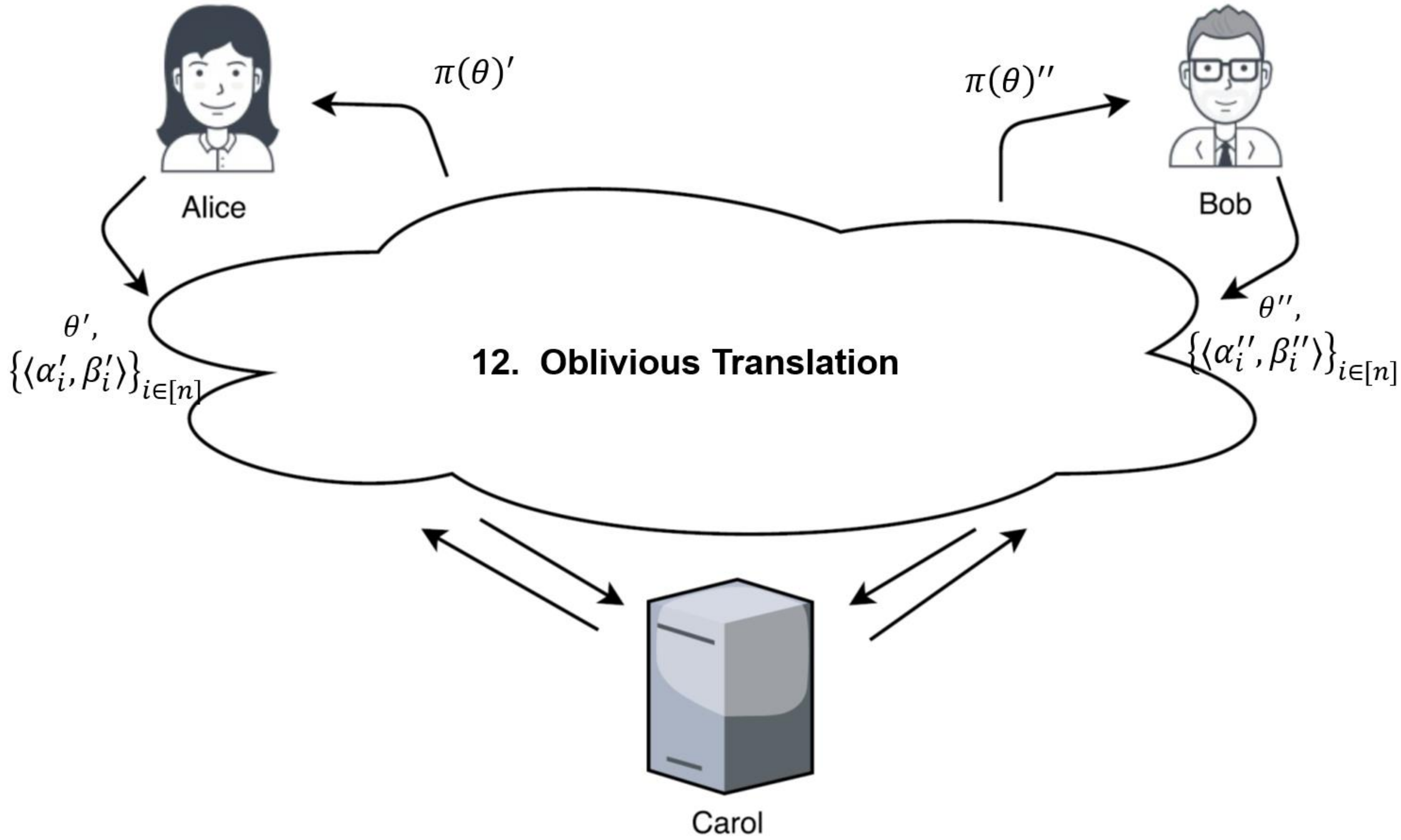


$((1-\beta)d)'$
$((1-\beta)d)''$
Alice
Bob
$(1-\beta)', d'$
6. Oblivious Multiplication
$(1-\beta)'', d''$
Carol


Alice
Bob
7. $(\beta c)' + ((1-\beta)d)'$
7. $(\beta c)'' + ((1-\beta)d)''$
Carol
8. Computers the Sum of two terms received from A and B to get:
$\beta c + (1-\beta)d = \begin{cases} c \text{ if } \beta = 1 \\ d \text{ if } \beta = 0 \end{cases}$
Note: The result in (8) is the address of next instruction to be executed.



$\pi_1(P)', \pi_2(D)'$
$\pi_1(P)'', \pi_2(D)''$
Alice
Bob
P' , D'
P'', D''
11. Oblivious Shuffle
Carol


Alice
Bob
Carol
$\pi(\theta)'$
$\pi(\theta)''$
θ',
$\{\langle \alpha_i', \beta_i' \rangle\}_{i \in [n]}$
θ'',
$\{\langle \alpha_i'', \beta_i'' \rangle\}_{i \in [n]}$
12. Oblivious Translation

Implementation

- Programming Language: C++.
- Three Modules: Alice, Bob running on remote hosts, Carol is an offline preprocessor.
- The online parties communicate with each other over a network via TCP\IP sockets.
- We have successfully implemented the complete protocol.
- From test runs, the running times are promising.

Q & A

THANKS! ☺