

## 1 Introduction

Consider the *Billionaire's Problem*: Larry and Sergey are both wealthy men. They want to design a protocol to find out whose net worth is higher, without having to reveal their net worths to each other. Since they are business partners, they trust each other to follow the protocol truthfully (i.e., to use their true worth as inputs and to follow the protocol instructions faithfully), but they still do not want the protocol to reveal more about their net worths than is absolutely necessary. Can this be achieved? More fundamentally, how do we define the notion of security for this goal?

Note that “total” privacy cannot be achieved, as they will each learn something new about the other’s wealth, namely, whether it is greater or less than his own. In some special cases, Larry can infer Sergey’s net worth *entirely* from just this single piece of knowledge. For example, if Larry’s net worth is \$1 and Sergey’s net worth is \$0, then when Larry learns that Sergey’s worth is less than his own, he can infer that Sergey’s net worth is exactly \$0 (we ignore the possibility of Sergey being in debt). However, this leakage of knowledge is *inherent in the task they are carrying out for these values*, and therefore should not be considered a deficiency of any particular protocol they might use.

Alternatively, Larry might start with some prior knowledge about Sergey’s wealth that might enable him to infer Sergey’s exact net worth from just knowing whose net worth is higher. For example, if Larry knows that Sergey’s net worth is either \$4 billion or \$5 billion, and his own worth is \$4.5 billion, then learning who is wealthier immediately reveals Sergey’s net worth to Larry. This also should not be considered a violation of our security goal.

In both of these examples, we must be content to let Larry learn whatever he can infer from the final result (i.e., who is wealthier) — *but we want him to learn nothing more than that!* So in defining security, we will aim to restrict the “*relative knowledge*” revealed by a protocol, versus what is learned from the outcome alone. Similarly to the setting of zero knowledge, we will do so using the notion of an efficient simulator that is given only the input and output of the party, and must simulate the entire view of that party in the protocol.

## 2 Secure Two-Party Computation

Here we formalize a model and security definition for the informal goals described above.

### 2.1 Model

We will consider a very simplified model that does not capture many real-world concerns, but is still rich enough to make the problem interesting and non-trivial.

1. There are two parties (more formally, two ppt algorithms)  $P_1$  and  $P_2$ , who have inputs  $x_1$  and  $x_2$  respectively, and wish to evaluate a public polynomial time-computable function  $f(\cdot, \cdot)$  on those inputs. For example, in the billionaire’s problem,  $f(x_1, x_2) = [x_1 > x_2]$ .

Without loss of generality, we may assume that  $f$  is a *deterministic* function that outputs a *single* value that is given to both parties. If we wish for  $P_1$  and  $P_2$  to receive the outputs of two possibly different deterministic functions  $f_1(\cdot, \cdot)$ ,  $f_2(\cdot, \cdot)$  (respectively), this can be emulated using a single function  $f'((x_1, r_1), (x_2, r_2)) = (f_1(x_1, x_2) \oplus r_1 \| f_2(x_1, x_2) \oplus r_2)$ , where each  $P_i$  augments its own input  $x_i$  by a uniformly random string  $r_i$  of appropriate length. Since  $r_1$  and  $r_2$  are chosen uniformly at random and are independent of everything else, the output  $f_1(x_1, x_2)$  is perfectly hidden from  $P_2$ , as is  $f_2(x_1, x_2)$  from  $P_1$ .

We can also evaluate a *randomized* function  $f$  by emulating it with a deterministic function (showing how to do this is one of your homework problems). However, security becomes quite a bit subtler to define in this case; see below.

2. We assume that the parties are *semi-honest*, often called “honest but curious.” That is, they run the protocol exactly as specified (no deviations, malicious or otherwise), but may try to learn as much as possible about the input of the other party from their views of the protocol. Hence, we want the view of each party not to leak more knowledge than necessary.
3. As usual, the view of a party  $P_i$  in an interaction with the other party on their inputs  $x_1, x_2$ , denoted  $\text{view}_{P_i}[P_1(x_1) \leftrightarrow P_2(x_2)]$ , consists of its input  $x_i$ , the random coins  $r_{P_i}$  used by  $P_i$ , and all the messages received from the other party. The final output of  $P_i$  is denoted  $\text{out}_{P_i}[P_1(x_1) \leftrightarrow P_2(x_2)]$ .

## 2.2 Security Definition

**Definition 2.1.** A pair of ppt machines  $(P_1, P_2)$  is a secure 2-party protocol (for static, semi-honest adversaries) for a deterministic polynomial time-computable function  $f(\cdot, \cdot)$  if the following properties hold:

1. *Completeness*: for all  $i \in \{1, 2\}$  and all  $x_1, x_2 \in \{0, 1\}^*$ , we have (with probability 1):

$$\text{out}_{P_i}[P_1(x_1) \leftrightarrow P_2(x_2)] = f(x_1, x_2).$$

2. *Privacy*: there exist nuppt simulators  $\mathcal{S}_1, \mathcal{S}_2$  such that for all  $x_1, x_2 \in \{0, 1\}^*$  and all  $i \in \{1, 2\}$ ,

$$\text{view}_{P_i}[P_1(x_1) \leftrightarrow P_2(x_2)] \stackrel{c}{\approx} \mathcal{S}_i(x_i, f(x_1, x_2)).$$

A few remarks are in order. First, privacy is *per-instance*: the only knowledge leaked to a party by the protocol on inputs  $x_1, x_2$  is whatever can be inferred (efficiently) from the party’s own input and the value of  $f(x_1, x_2)$ . For example, if the inputs are such that  $f(x_1, x_2)$  reveals nothing at all, then the execution of the protocol on those inputs should also reveal nothing; conversely, if the output reveals everything about both parties’ inputs, then the protocol is allowed to leak everything as well. Second, any “prior knowledge” that the parties have about each others’ inputs is captured by the non-uniformity of the simulator (and implicit distinguisher) in the definition of privacy.

## 2.3 Definition for Randomized Functions

Definition 2.1 is relatively straightforward due to the simplicities of our model, in particular, the deterministic nature of  $f$ . We briefly discuss some of the issues that arise in defining security for randomized functions. First, how should completeness be defined? It no longer makes sense to demand that  $\text{out}_{P_1} = f(x_1, x_2)$ , since we now have a random variable  $f(x_1, x_2; r)$  over the choice of  $r$  (which neither party should be able to influence). Instead, we want that both  $\text{out}_{P_1}$  and  $\text{out}_{P_2}$  in a *single* execution of the protocol are *simultaneously* distributed as  $f(x_1, x_2; r)$  for *the same* random  $r$ . This is so that the protocol between  $P_1$  and  $P_2$  has the effect of emulating a single, consistent randomized evaluation of  $f$ . Formally, we want that for all  $x_1, x_2$ ,

$$(\text{out}_{P_1}, \text{out}_{P_2})[P_1(x_1) \leftrightarrow P_2(x_2)] \stackrel{c}{\approx} (f(x_1, x_2; r), f(x_1, x_2; r)),$$

where  $r$  is uniformly random and the same in both appearances of  $f$ .

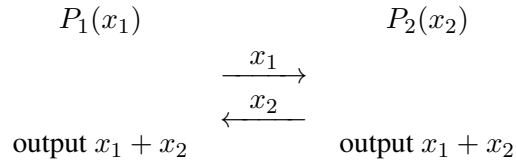
The next natural question is how to define *privacy* against a semi-honest party. Again, simultaneity of the respective views of  $P_1$  and  $P_2$  is an important issue, and is even more subtle to get right. It turns out that the proper way of addressing all these concerns is to define correctness and privacy *all together* by comparing two *joint* distributions: the “real world” distribution of the parties’ outputs and the semi-honest party’s view, versus the “ideal world” distribution of the function output and simulated view (again, for a *single* randomized evaluation of  $f$ ). Formally, we require that there exist nuppt simulators  $\mathcal{S}_1, \mathcal{S}_2$  such that for all  $x_1, x_2$  and all  $i \in \{1, 2\}$ ,

$$(\text{out}_{P_1}, \text{out}_{P_2}, \text{view}_{P_i})[P_1(x_1) \leftrightarrow P_2(x_2)] \stackrel{c}{\approx} (f(x_1, x_2; r), f(x_1, x_2; r), \mathcal{S}_i(x_i, f(x_1, x_2; r))).$$

Note that the above condition automatically implies the correctness condition above, so it is the only one needed to prove security.

## 2.4 Secure Protocol for Addition

As a brief test case, we consider a contrived protocol for evaluating the addition function  $f(x_1, x_2) = x_1 + x_2$ .



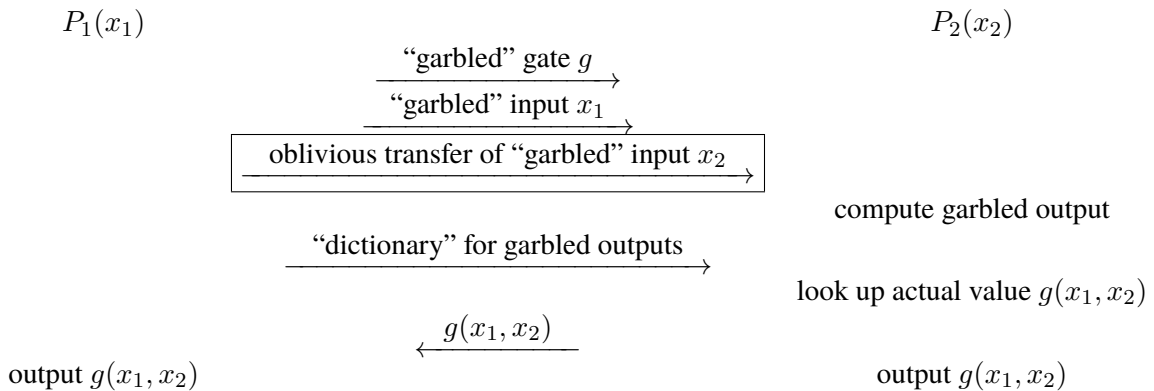
Clearly the protocol is complete. For privacy, since  $P_1$  is entitled to know the value of  $x_1 + x_2$ , and also already knows  $x_1$ , he can trivially infer the value of  $x_2$ . Formally, we can give a simulator  $\mathcal{S}_1(x_1, s = f(x_1, x_2) = x_1 + x_2)$  that just outputs the view consisting of input  $x_1$ , empty randomness, and a single message  $s - x_1$  coming from  $P_2$ . Clearly, this view is identical to  $P_1$ ’s view in the real protocol.

## 3 Secure Protocol for Arbitrary Circuits

We now describe a protocol, originally described by Yao, for evaluating an *arbitrary* function  $f$  represented as a boolean (logical) circuit. We describe the basic idea for just a single logic gate, and then outline how it generalizes to arbitrary circuits.

Let  $g : \{0, 1\} \times \{0, 1\} \rightarrow \{0, 1\}$  be an arbitrary logic gate on two input bits (e.g., the NAND function). Party  $P_1$  holds the first input bit  $x_1 \in \{0, 1\}$  and party  $P_2$  holds the second input bit  $x_2 \in \{0, 1\}$ . Together they wish to compute  $g(x_1, x_2)$  securely, in the sense of Definition 2.1.

At a high level, the protocol works like this:



For intuition, the crucial points for security are:

- $P_2$  never sees  $x_1$  in an “ungarbled” form, so  $P_2$  learning nothing about  $x_1$ .
- By the security of the “oblivious transfer” sub-protocol (described below),  $P_1$  learns nothing about  $x_2$ .
- Using the garbled inputs  $x_1, x_2$  with the garbled gate,  $P_2$  can “obliviously” compute the garbled output for *only* the correct value of  $g(x_1, x_2)$ .

Concretely, these ideas are implemented using basic symmetric-key encryption. The idea is the following: each “wire” of the gate (the two inputs and one output) is associated with a pair of random symmetric encryption keys, chosen by  $P_1$ ; the two keys correspond to the two possible “values” (0 or 1) that the wire can take. For  $i = 1, 2$ , let  $k_0^i, k_1^i$  be the keys corresponding to the input wire  $x_i$ , and let  $k_0^o, k_1^o$  be the keys corresponding to the output wire. The “garbled circuit” that  $P_1$  sends to  $P_2$  is a table of four doubly encrypted values, presented in a *random* order:

$\text{Enc}_{k_0^1}(\text{Enc}_{k_0^2}(k_{g(0,0)}^o))$
$\text{Enc}_{k_0^1}(\text{Enc}_{k_1^2}(k_{g(0,1)}^o))$
$\text{Enc}_{k_1^1}(\text{Enc}_{k_0^2}(k_{g(1,0)}^o))$
$\text{Enc}_{k_1^1}(\text{Enc}_{k_1^2}(k_{g(1,1)}^o))$

Observe that if  $P_2$  knows (say)  $k_0^1$  and  $k_1^2$ , i.e., the keys corresponding to inputs  $x_1 = 0$  and  $x_2 = 1$ , then  $P_1$  can decrypt  $k_{g(0,1)}^o$ , the key corresponding to the output value of the gate, *but none of the other entries!* (Note that this requires the encryption scheme to satisfy some simple properties, such as the ability to detect when a ciphertext has decrypted successfully. These properties are easy to obtain.) The random order of the table prevents  $P_1$  from learning the “meaning” of the keys that it knows, otherwise this information would be leaked by which of the table entries decrypt properly. In conclusion, knowing exactly one key for each input wire allows  $P_2$  to learn exactly one key (the correct one) for the output wire, without learning the meanings of any of the keys.

The only remaining question is how  $P_2$  obtains the right keys for the input wires. For  $x_1$ , this is easy:  $P_1$  just sends  $P_2$  the key  $k_{x_1}^1$  corresponding to its input bit  $x_1$ . Note that  $P_2$  learns nothing about  $x_1$  from this. Next,  $P_1$  and  $P_2$  run an “oblivious transfer” protocol (described in the next subsection) which allows  $P_2$  to learn  $k_{x_2}^2$ , and *only*  $k_{x_2}^2$ , without revealing anything about the value of  $x_2$  to  $P_1$ .

Finally,  $P_1$  tells  $P_2$  the “meanings” of the two possible output keys  $k_0^o, k_1^o$ , which reveals to  $P_2$  the value of  $g(x_1, x_2)$ . Then  $P_2$  sends this value to  $P_1$  as well (recall that both parties are semi-honest, so neither will lie).

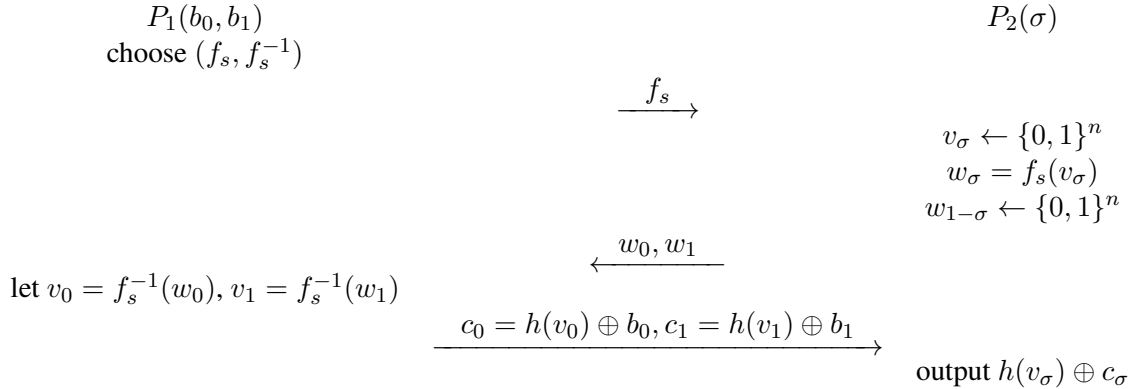
For more complex circuits  $f$ , the protocol generalizes in a straightforward manner:  $P_1$  chooses two keys for every wire in the circuit, and constructs a garbled table for each gate, using the appropriate keys for the input and output wires.  $P_2$  can compute the garbled gates iteratively, while remaining oblivious to the meanings of the intermediate wires. Then  $P_1$  finally reveals the meanings of just the output wires.

A proof of security for this construction is beyond the scope of this lecture, but contains no surprises; see the paper by Lindell and Pinkas for a full rigorous proof. The key point is that a simulator can construct garbled gates that *always* result in the same key being decrypted (irrespective of which inputs keys were used), thus allowing the simulator to “force”  $P_2$  to output the correct value  $f(x_1, x_2)$ . Security of the symmetric encryption scheme prevents  $P_2$  from detecting these malformed garbled gates, since  $P_2$  can only decrypt one entry from each gate.

## 4 Oblivious Transfer

We conclude by describing how to perform an oblivious transfer between the two parties. We will consider the specific form of oblivious transfer that is required to complete our protocol:  $P_1$  is holding two bits  $b_0, b_1$  and wants to transfer *exactly one* of them to  $P_2$ , according to  $P_2$ 's choice bit  $\sigma = x_2$ , while learning nothing about which one was received. (To transfer entire keys  $k_0, k_1 \in \{0, 1\}^n$ , the parties can just run the protocol  $n$  times, using the same choice bit  $\sigma$  and the corresponding pairs of bits from  $k_0, k_1$ ).

Our protocol relies on a family of trapdoor permutations  $\{f_s: \{0, 1\}^n \rightarrow \{0, 1\}^n\}$  with hard-core predicate  $h: \{0, 1\}^n \rightarrow \{0, 1\}$ .



In words,  $P_1$  picks a random function  $f_s$  (with trapdoor) from the family and sends it to  $P_2$ . Then  $P_2$  chooses uniformly random  $w_0, w_1 \in \{0, 1\}^n$  so that it knows the preimage of *only*  $w_\sigma$ , and sends these to  $P_1$ . (Observe that this reveals no information about  $\sigma$  to  $P_1$  because  $w_0, w_1$  are uniform and independent.) Next,  $P_1$  encrypts its two bits  $b_0, b_1$  using the hard-core predicate  $h$  on the preimages of  $w_0, w_1$ , respectively. Finally  $P_2$ , knowing the preimage  $w_\sigma$ , can decrypt  $b_\sigma$ , but it learns nothing about  $b_{1-\sigma}$  due to the hardness of  $h$ . Note that the protocol crucially relies on the fact that  $P_2$  is semi-honest, otherwise it could choose  $w_0, w_1$  so that it knew both preimages. (A full, formal proof of security for this protocol is not too hard, and is a worthwhile exercise.)