Today we will take a break from encryption and consider an interesting application of some of the things we learned so far. Sometime in the early 1980s Andrew Yao came up with this example. Suppose Alice and Bob are CEOs of two big companies and they want to know who has the higher salary. However they do not want to reveal the amount of salary to one another, or to anyone else – they just want to know who is getting paid more. What are they supposed to do?

This example is a special case of the general problem of private two-party computation. Alice and Bob each hold their private inputs x and y respectively, and they want to compute some function $f(x, y)$ of these inputs. However, beyond discovering the value of the function, they want to keep their inputs hidden from one another and from the rest of the world.

As usual, in order to obtain any kind of security requirement, we will need both the functionality requirement and the security requirement of the task at hand. Describing the functionality requirement for private two-party computation is going to be a bit more difficult. In the cryptographic tasks that we have seen so far (encryption, authentication), the exchange of messages between Alice and Bob was *non-interactive*: One of the parties sends a single message, and the functionality requirement specifies how the other party should behave upon observing this message. Fortunately, non-interactive exchange was sufficient to achieve these tasks. In contrast, private two-party computation requires an interactive exchange of messages between the two parties. This leads us to the definition of interactive protocols.

1 Two-party interactive protocols

Today we will only consider two-party interactive protocols. Such a protocol consists of two randomized algorithms A and B. The algorithms obtain private inputs x and y respectively and exchange a sequence of messages that is determined by their inputs as well as their (internal) randomness. The exchange of messages proceeds in rounds: One of the parties sends a message, the other one observes the message and sends back a response, and so on, until some point in time when each of the parties halts with an output written somewhere in its memory. Without loss of generality, we will assume that Alice starts the interaction.

Formally, an *interactive algorithm* takes three inputs: A private input $x \in \{0,1\}^*$, a round number $i \in \mathbb{N}$, and a sequence $\alpha_1, \ldots, \alpha_{i-1} \in \{0,1\}^*$ of messages exchanged so far, and produces either a next-message response $\alpha_i \in \{0,1\}^*$ or the special message `halt`, after which the algorithm terminates. The output of A is the value written in some special place in its memory just before it produces the `halt` message.

An *interactive protocol* is a pair of randomized interactive algorithms (A, B). The *transcript* of (A, B) on inputs x and y is the sequence of messages exchanged between A and B on inputs x and y respectively. The transcript is a random variable that depends on the internal randomness of A and B. The *view* of algorithm A in (A, B) on inputs x and y consists of the transcript together with the internal randomness of A. We write $(A(x), B(y))$ to denote the pair of outputs produced by the interactive protocol (A, B) on inputs x and y.

We can now define two-party computation:

Definition 1. Let $f, g\colon \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^*$ be a pair of functions. A *two-party computation* of (f, g) is an interactive protocol (A, B) so that for every pair of inputs $x, y \in \{0,1\}^*$, $(A(x), B(y)) = (f(x, y), g(x, y))$ (with probability one over the internal randomness of A and B).

This definition is a bit more general than the scenario we considered. Here there are two, possibly different, functions f and g in whose values Alice and Bob may be interested in.

2 Security of two-party computation

One way to do a two-party computation is for Alice and Bob to exchange their inputs and compute $f(x, y)$ and $g(x, y)$, respectively. This is a perfectly valid protocol, but it reveals all sorts of information not only to Alice and Bob, but also to any observer.

So should we say that a two-party computation is secure? Let us start with the simpler scenario of security against "passive" adversaries. We will assume that Alice and Bob communicate over a two-way public but authenticated channel: That is, Bob knows that Alice's messages indeed originate from Alice, and vice-versa. The bare minimum we would want is that the messages exchanged between Alice and Bob remain private relative to an observer. We can achieve this using public-key encryption.

However, if we look back at the scenario of Alice and Bob wanting to find out who has the higher salary, it becomes apparent that we want more: Not only do Alice and Bob want privacy relative to an observer, but they also want privacy relative to one another. In other words, we want that at the end of the protocol execution

- Alice cannot learn anything about Bob's input y beyond what is implicitly revealed in her output value $f(x, y)$;
- Bob cannot learn anything about Alice's input x beyond what is implicitly revealed in his output value $g(x, y)$.

Here is what we mean. Suppose that $f(x, y)$ is the equality predicate, that is $f(x, y) = 1$ when $x = y$ and 0 otherwise. If Alice has input $x = 128957$ and at the end of the protocol she finds out that $f(x, y) = 1$, then she will inevitably learn what Bob's input is. However, if instead she finds out that $f(x, y) = 0$, she will have no idea which input Bob holds beyond the fact that it is not 128957.

How do we define this notion of security? From our experience so far, a reasonable guess would be to ask for some sort of "transcript indistinguishability". Namely, for every input x of Alice and every pair of inputs y_0, y_1 to Bob such that $f(x, y_0) = f(x, y_1)$, we would want to say that Alice's view on input pair (x, y_0) should be computationally indistinguishable from Alice's view on input pair (x, y_1), and define a similar condition for Bob.

The simulation paradigm I think such a definition would make sense, but for reasons that will (hopefully) become apparent in the next lecture, it is better to work directly with a semantic notion of security. Recall the intuition behind semantic security: All the knowledge an observer can gain after participating in an interaction, he can simulate (with comparable efficiency) all by himself.

What does Alice learn after interacting with Bob? Apart from the output value $f(x, y)$, which she is supposed to learn, she may also gain some unintended knowledge by "data mining" her view of the interaction. From her perspective, what she sees is a sample from a distribution on views, where each possible view comes from a different setting of Bob's internal randomness. If she can reproduce this distribution on views all by herself, then we can sensibly say that she has not learned anything (beyong the value $f(x, y)$ by interacting with Bob.

So here is our attempt at a definition: We want to say that a two-party computation (A, B) is "secure" against Alice if Alice's view (as a probability distribution) can be simulated given only access to x (her input) and $f(x, y)$ (the output that she learns at the end of the protocol). But what does it mean to simulate a probability distribution? Ideally, we would want to say that the distribution simulated by Alice is the same as the distribution observed after her interaction with Bob. But this "perfect security" requirement is sometimes impossible to achieve, so we relax it and require that the two distributions be only *computationally* indistinguishable.

Definition 2. Two random variables X and Y (taking values in $\{0,1\}^m$) are (s, ε) *computationally indistinguishable* if for every circuit D of size s,

$$|\Pr[D(X) = 1] - \Pr[D(Y) = 1]| \leq \varepsilon.$$

We have already seen an important example of this definition: $G\colon \{0,1\}^n \to \{0,1\}^m$ is a pseudo-random generator if and only if the random variables $G(U_n)$ and U_m are computationally indistinguishable, where U_n and U_m are the uniform distributions over $\{0,1\}^n$ and $\{0,1\}^m$, respectively.

Definition of semantic security We can now give a "semantic security" definition of secure two-party computation.

Definition 3. A two-party computation (A, B) of (f, g) is (s, ε)-*secure against honest-but-curious parties* with view-simulating (randomized) algorithms (S_A, S_B) if for every pair of inputs (x, y)

- The random variable $S_A(x, f(x, y))$ is (s, ε) computationally indistinguishable from the view of A in the interactive protocol (A, B) on inputs x, y, and
- The random variable $S_B(y, f(x, y))$ is (s, ε) computationally indistinguishable from the view of B in the interactive protocol (A, B) on inputs x, y.

The running time of S_A and S_B is meant to be comparable to the running times of A and B; remember the intuition that whatever "knowledge" Alice can learn by observing the view of her algorithm A, she can simulate by running $S_A(x, f(x, y))$.

This notion of security is actually very loose. Notice that we only require "security" when Alice and Bob follow the rules of the prescribed protocol. If one of the parties tries to cheat by doing something unexpected, the other party may inadvertently reveal its whole input without ever knowing that this happened! We will illustrate this issue shortly when we give some examples of two-party protocols.

However, it turns out that although the honest-but-curious threat model may not be terribly realistic, it is extremely useful. In the next few lectures we will introduce a general technique that converts any protocol that is secure against honest-but-curious parties and converts it into a

protocol with the same functionality that is secure against much more realistic threats. We can make an analogy with private-key encryption: While encryption for a single message is not that useful in practice, it served us well as a stepping stone for designing much more secure schemes.

3 Oblivious transfer

As a warm-up, let's consider a two-party protocol for a very simple functionality. This should give us some practice with the definition. Let us consider the following functionality (f, g):

$$f(x, y) = \perp \text{ (nothing)} \quad \text{and} \quad g(x, y) = (x_1 + \cdots + x_n) + (y_1 + \cdots + y_n).$$

Here, $x = x_1 \dots x_n, y = y_1 \dots y_n$, and $+$ is XOR as usual. Consider the following protocol (A, B):

A: Send $x_1 + \cdots + x_n$ and halt.

B: Upon receiving a, output $a + y_1 + \cdots + y_n$ and halt.

Alice's view in this protocol is empty, so it can be trivially simulated. Bob's view is simply $x_1 + \cdots + x_n$. From his input y and his output $g(x, y)$, he can simulate his view by computing

$$S_b(y, g(x, y)) = y_1 + \cdots + y_n + g(x, y).$$

In this case, there is no randomness involved so the simulation of the protocol is straightforward. However these kinds of protocols exist only for very few (not terribly interesting) functionalities.

To go further, it is helpful to look at a special case – but not an obvious one. It is a functionality called *oblivious transfer*. Alice's input consists of n bits $x_1, \dots, x_n$, while Bob's input is a number i between 1 and n. At the end of the protocol, Alice learns nothing, while Bob learns the value x_b.

$$f(x, b) = \perp \quad \text{and} \quad g(x, b) = x_b.$$

What is supposed to happen in an honest-but-curious protocol for oblivious transfer is that Bob learns the value x_i but none of the values of the other bits, and Alice has no idea which value Bob has learned (beyond the fact that he learned exactly one of them)!

Here is a "physical" realization of this protocol. Alice has n boxes, each with a different key-lock pair. She puts each of the bits $x_1, \dots, x_n$ in its corresponding box, locks the box, and sends all n *keys* to Bob. Bob keeps key b and destroys all the other keys. Alice then sends the n boxes to Bob. Bob can then find out the value of x_b, but at this point he cannot learn anything about what is in the other boxes.

If Bob behaves the way he is supposed to (namely, he gives his best at destroying the other $n-1$ keys) and the locks are good enough, then this protocol should indeed be secure: After the interaction, Bob finds out the value x_i and nothing else, while Alice finds out nothing about Bob's value i. A dishonest Bob can keep all the keys instead of destroying them and find out all of Alice's hidden bits; but this is a worry for the next lecture.

What we need to worry about now is how we can implement this protocol in the digital world. The issue is that, unlike keys, bits of information cannot be destroyed. Once a bit makes it into Bob's view of the interaction, it stays there forever.

However, one thing that is (presumably) possible in the digital world but not in the physical world is public-key cryptography! To describe the protocol, we need a family of trapdoor permutations f_{PK} (with key generation Gen and trapdoor inversion Inv) and respective hardcore bits h_{PK}. For simplicity, we describe the protocol for $n = 2$.

The oblivious transfer protocol Recall that Alice has input $x_0x_1 \in \{0,1\}^2$ and Bob has input $b \in \{0,1\}$.

A: Using Gen, produce a pair of keys (SK, PK) and send PK to Bob.

B: Choose two random strings z_0 and z_1 from the domain of f_{PK}.

- If $b = 0$, send the pair $(f_{PK}(z_0), z_1)$ to Alice.
- If $b = 1$, send the pair $(z_0, f_{PK}(z_1))$ to Alice.

A: Upon receiving (u_0, u_1), send the pair $(h_{PK}(Inv(SK, u_0)) + x_0, h_{PK}(Inv(SK, u_1)) + x_1)$ to Bob and halt.

B: Upon receiving (a_0, a_1), output $h_{PK}(z_b) + a_b$ and halt.

It is easy to see that the protocol computes the desired functionality. We now show security:

Claim 4. *Suppose h_{PK} is an (s, ε) hardcore bit family for f_{PK}. Then the oblivious transfer protocol is $(s - O(t), \varepsilon)$-secure against honest-but-curious parties with the view-simulating algorithms S_A and S_B given in the proof, where t is the circuit size of computing Gen, f_{PK}, and Inv.*

Proof. Let us first look at Alice's view. Alice's view is $(SK, PK, f_{PK}(z_0), z_1)$ when $b = 0$ and $(SK, PK, z_0, f_{PK}(z_1))$ when $b = 1$. Both of these distributions are identical (namely, $(\infty, 0)$-indistinguishable) to the output produced by the following simulator:

> $S_A(x_0, x_1, \perp)$: Run Gen to obtain a pair (SK, PK), choose u_0, u_1 at random from the domain of f_{PK}, and output (SK, PK, u_0, u_1).

Let's now look at Bob's view. Bob's view consists of his internal randomness z_0, z_1 and the messages received from Alice. When $b = 0$, the message received in the first round is PK, while the message received in the second round is $h_{PK}(z_0) + x_0, h_{PK}(Inv(SK, z_1)) + x_1$, so Bob needs to simulate the distribution

$$(z_0, z_1, PK, h_{PK}(z_0) + x_0, h_{PK}(Inv(SK, z_1)) + x_1).$$

Since f_{PK} is a permutation, this distribution is identical to

$$(z_0, f_{PK}(z_1), PK, h_{PK}(z_0) + x_0, h_{PK}(z_1) + x_1).$$

The first four elements of the distribution are easy to simulate, but the last one looks difficult because Bob doesn't know x_1. But $h_{PK}(z_1)$ is a hardcore bit for $f_{PK}(z_1)$, so it should look random to Bob! This suggests the following simulator:

$S_B(b, x_b)$: Run *Gen* to obtain a pair (SK, PK), choose z_0, z_1 at random from the domain of f_{PK}, and choose a random bit $r \sim \{0,1\}$. Output

$$\begin{array}{ll} (z_0, f_{PK}(z_1), PK, h_{PK}(z_0) + x_b, r) & \text{if } b = 0, \\ (f_{PK}(z_0), z_1, PK, r, h_{PK}(z_1) + x_b) & \text{if } b = 1. \end{array}$$

Suppose for contradiction that Bob's view and $S_B(b, x_b)$ are $(s - O(t), \varepsilon)$-computationally distinguishable, say for $b = 0$. Then the distributions $(f_{PK}(z_1), h_{PK}(z_1))$ and $(f_{PK}(z_1), r)$ are $(s - O(1), \varepsilon)$-computationally distinguishable (by the usual argument), and so h_{PK} is not an (s, ε) hardcore bit for f_{PK}. □

Notice that this protocol is blatantly insecure if Bob does not follow the prescribed steps. If in step 2 Bob sends the pair $(f_{PK}(z_0), f_{PK}(z_1)$, then the view of Alice will be distributed identically as when Bob is honest, but Bob will be able to learn both the values x_0 and x_1 at the end of the protocol. Miraculously, in the next few lectures we will show a general technique for "upgrade" protocols so they become resilient to this type of attack!

Before we move on, let's point out one technical inaccuracy that was lost in our analysis. To illustrate it, suppose that the trapdoor permutation used in the protocol is Rabin's permutation $f(x) = x^2$ on the set of quadratic residues Q_n, where n is a product of primes of the required form. In the second step of the protocol, Bob needs to choose random elements z_0, z_1 from Q_n. However, the only way we know how to choose z_0, z_1 at random from Q_n is to first choose two random elements $v_0, v_1 \sim \mathbb{Z}_n^*$ and set $z_0 = v_0^2, z_1 = v_1^2$. Then Bob's view needs to include v_0 and v_1 and not merely z_0 and z_1. In that case, it becomes impossible to obtain the simulator S_B without factoring n first!

To get around this problem, it is common to impose an additional requirement on the trapdoor permutation. In the original definition, we required that when given PK and a random z in the domain of f_{PK}, it is hard to find an inverse of $f_{PK}(z)$. Here, we want to make the stronger requirement that given PK, a random z from the domain of f_{PK}, *and the randomness used to generate* z, it is still hard to find an inverse of $f_{PK}(z)$. A trapdoor permutation that satisfies this additional requirement is called an *enhanced trapdoor permutation*. Rabin't permutation $f(x) = x^2$ is *not* an enhanced trapdoor permutation on Q_n; however, the function $f(x) = x^4$ is an enhanced trapdoor permutation, provided factoring public keys obtained by *Gen* is hard.