In the last three lectures we introduced several ingredients that go into the design of secure two-party computation protocols. Let us recall what they are:

- Secure two-party computation in the honest-but-curious model: For every pair of functionalities $(f(x,y), g(x,y))$ we gave a pair of interactive algorithms A, B, so that the output of the protocol $(A(x), B(y))$ equals $(f(x,y), g(x,y))$, and the view of either party in this protocol can be simulated given only its input and output.
- A bit commitment protocol: This protocol gives a way for Bob to commit to a value x to Alice in a way that is hiding (Alice cannot distinguish commitments to different values) and binding (Bob can only decommit to the value he has previously committed to).
- A coin-tossing protocol: This protocol gives a way for Alice and Bob to agree on a random string.
- Zero-knowledge proofs for any proof relation: In this protocol, Bob wants to convince Alice that a statement x is true, but without revealing any additional information. In addition to x, Bob has a proof y (which we model as membership in some proof relation R). At the end of the interaction, Alice is convinced that x is true, but does not gain information about y (or anything else).

Today we will sketch how these elements go into the construction for secure two-party computation for arbitrary functionalities with malicious parties. The construction will require an additional feature of (certain) zero-knowledge proofs called "knowledge extractability". But before we go into it, we need to give a definition of secure two-party computation.

1 A definition of secure two-party computation

Before we attempt a definition of secure two-party computation, let us imagine what things can go wrong when we try to achieve a "secure" implementation of a two-party functionality. Recall that in a two-party functionality, Alice and Bob have their private inputs x and y and their goal is to obtain the values $f(x,y)$ and $g(x,y)$.

In an ideal definition of secure two-party computation, we would like to make as few assumptions as possible about how a dishonest party would act in a protocol for computing (f,g). Essentially, we would like to say that a dishonest party should be able to do whatever they want. By allowing a dishonest party to behave in an arbitrary way, however, we may affect the *functionality* of the computation. A dishonest Alice may for example provide a "fake" input x^* to the protocol, or simply refuse to participate in the protocol altogether. However, we want to say that even these functionally devious behaviors do not affect the *security* of the protocol: As long as one of the two parties is honest, no matter what the other party does, the dishonest party should not be able to deduce any private information by running the protocol.

For simplicity let us first consider asymmetric functionalities of the type $(x,y) \to (\perp, g(x,y))$ where only Bob gets an output from the computation. One possibility in such computations is that Alice

simply refuses to participate in the protocol (since she gets nothing out of it anyway). However refusing to participate is an observable action that may trigger some punishment in the real world, so she may want to avoid it.

Another possibility is that she participates in the protocol but does not play by the rules, with the hope of finding out some private information; perhaps she engages in a couple of rounds of interaction, at which point she finds out some of Bob's private information and she decides to abort. Our definition will rule out this possibility: If Alice aborts at any point during the protocol, Bob will not get his output, but the privacy of his input will not be compromised.

Here is a more subtle attack that can be done from Bob's side. After Alice sends the first message, Bob replaces his actual input x with some information that Alice revealed in her first message. Although by itself Alice's first message may not reveal anything unintended, it is possible that by correlating his private input with it, the output of the protocol would reveal some of Alice's private information.

A useful way to decouple functionality attacks from security attacks is by introducing an "ideal protocol" of secure computation, in which Alice and Bob can count on a trusted authority T to do the secure computation for them. In an ideally secure implementation of the computation $(x, y) \to (\perp, g(x, y))$ this is what we may expect to happen:

The ideal protocol $(\tilde{A}, \tilde{B})$**.** On inputs x for Alice and y for Bob:

1. Alice submits x privately to the trusted party T. At the same time, Bob submits y privately to T.
2. Upon receiving x and y, T performs the computation $g(x, y)$, privately reveals the output to Bob, and sends the message $\perp$ to Alice. If Alice did not send an input in the first round, T sends the message `Alice aborted` to Bob. If Bob did not send an input, T forwards the message `Bob aborted` to Alice.
3. Alice and Bob (privately) output the message sent to them by T.

In this ideal protocol, Alice and Bob are allowed to (1) refuse participating in the protocol and (2) submit inputs to the trusted party that are different from their actual inputs. However, these are essentially the only "attacks" they can perform. If they refuse to participate, their refusal can be detected (by the trusted party), and in that case no party gets any information besides the fact that the other party aborted. On the other hand, if they both decide to participate (by submitting inputs x^* and y^*), then each party gets exactly what it is supposed to – Alice gets nothing while Bob gets $g(x^*, y^*)$.

Our definition of security will require that whatever attack Alice can mount in the real protocol against an honest Bob can be emulated by an Alice in the ideal protocol. In particular, it implies that (1) Privacy cannot be violated even if a party decides to abort or misbehave (send an invalid message) in the middle and (2) The inputs that Alice and Bob choose to run the protocol on are independent of one another (more precisely, if a party violates this condition it will be forced to abort the protocol).

The definition of security is simulation-based: It says that whatever information a malicious Bob B^* can obtain based on his view in the execution of the real protocol can be simulated by a malicious

Bob $\tilde{B}^*$ in the ideal model that either aborts, or chooses some input y^* and makes a decision based on y^* and the output $g(x, y^*)$ provided by the trusted party, and similarly for Alice.

Definition 1. Let $(\tilde{A}, \tilde{B})$ be the ideal protocol for the functionality $(x, y) \to (\perp, g(x, y))$. A pair of interactive algorithms (A, B) is an (s, ε) *secure implementation* of $(\tilde{A}, \tilde{B})$ with simulation overhead $oh(\cdot)$ if

- **Functionality:** The protocol $(A(x), B(y))$ computes $(\perp, g(x, y))$.
- **Security for Alice:** For every interactive algorithm B^* there exists an interactive algorithm $\tilde{B}^*$ so that for every input x, the distributions on (pairs of private) outputs of $(\tilde{A}(x), \tilde{B}^*)$ and $(A(x), B^*)$ are (s, ε) computationally indistinguishable, where the running time of $\tilde{B}^*$ is at most oh of the running time of B^*.
- **Security for Bob:** For every interactive algorithm A^* there exists an interactive algorithm $\tilde{A}^*$ so that for every input y, the distributions on (pairs of private) outputs of $(\tilde{A}^*, \tilde{B}(y))$ and $(A^*, B(y))$ are (s, ε) computationally indistinguishable, where the running time of $\tilde{B}^*$ is at most oh of the running time of B^*.

Let us compare this definition with the simulation-based definitions of security that we used to define security in the honest-but curious model, as well as the zero-knowledge property of proofs. Here, too, we would like to have a similar definition: We want to say that "whatever Alice can learn by observing the interaction in the actual protocol, she can also simulate in the ideal protocol". However, we cannot simply compare the views of the interactive algorithms A^* and $\tilde{A}^*$ representing Alice in the two protocols, as these views are syntactically different. Instead what we say is this: Based on her view, the interactive algorithm A^* can compute any output she wants in the real protocol. Our security requirement says that this output can be emulated in the ideal protocol (where Alice can only deviate either by aborting or choosing an input x^* different from x), even when conditioned on (honest) Bob's output.

This definition is quite strong and tricky to achieve even for seemingly trivial functionalities. As an example, let's look at a functionality of the type $(x, \perp) \to (\perp, g(x))$, where $g\colon \{0,1\}^n \to \{0,1\}^n$ is some function. A trivial protocol for this functionality is for Alice to simply send the value $g(x)$ to Bob, and for Bob to output this value, while Alice outputs $\perp$. However, this protocol may not be a secure implementation of the ideal protocol.

To see why, consider a cheating Alice A^* that chooses a random $r \in \{0,1\}^n$, sends r to Bob, and privately outputs r. To prove this protocol is secure, we need to come up with an interactive algorithm $\tilde{A}^*$, which includes an input x^* that Alice submits to the trusted party, so that $(\tilde{A}^*, g(x^*))$ is computationally indistinguishable from (A^*, r). In particular, $g(x^*)$ must be computationally indistinguishable from r. In certain cases – for example if g is the zero function – this is certainly impossible to achieve. But even if g is surjective (i.e., every r is a possible image of g), while in principle it should always be possible to find an x^* so that $g(x^*)$ is computationally indistinguishable from a random string, it may be computationally very expensive to do so, so it is not clear how to achieve a secure implementation with reasonably small simulation overhead.

2 Knowledge extractors

It seems that in order to design a secure implementation of the functionality $(x, \perp) \to (\perp, g(x))$, we need to hold Alice accountable that the value she claims for $g(x)$ was actually obtained by applying g to her input x. Since her input x is private, it is unavoidable (and allowed in the ideal protocol) that she produced Bob's output by applying g to some other input x^*. We have to ensure that this is the only way in which she can cheat.

A solution that comes to mind is zero-knowledge proofs. Consider the proof relation R_g consisting of those pairs (y, x) such that $y = f(x)$. After Alice sends the value $g(x)$, Bob may ask Alice to certify (in zero-knowledge) that $g(x)$ has a proof in R_g. However, in case $g(x)$ is surjective, this statement is vacuous: There certainly exists some x^* such that $(g(x), x^*) \in R_g$, so a zero-knowledge proof (or a proof of any kind) does not seem to help at all!

The issue here is somewhat different: In order to simulate a cheating Alice A^* in the real protocol by a cheating Alice $\tilde{A}^*$ in the ideal protocol, Bob needs to be convinced that Alice obtained the value $g(x)$ by indeed applying g to some input x; or in other words, that Alice "knows" x. But what does it mean for an interactive algorithm to know something? It seems that a precise definition of knowledge, in this context, is out of the question.

To explain it, let us revisit our discarded idea of realizing the implementation of realizing this functionality via a "zero-knowledge proof". Although the guarantee provided by zero-knowledge proofs seems inadequate for our task, let us look at an implementation of our zero-knowledge proof for the task at hand anyway:

Image transmission protocol On input x for Alice:

A: Send the value $g(x)$ to Bob.

A, B: Engage in the "zero-knowledge protocol" for statement $g(x)$ (available to Bob) and proof x (available to Alice). Specifically, repeat the following $4m \log(1/\varepsilon)$ times, where m is the number of edges in G below:

- A: Convert the pair $(g(x), x)$ into (G, col) where G is a graph and col is a 3-coloring. Send commitments to all the colors $col(v)$ to Bob.
- B: Convert the value r (supposedly $g(x)$) into a graph G'. Choose a random edge (u, v) of G' and ask Alice to open the commitments to $col(u)$ and $col(v)$.
- A: Send Bob the keys to the respective commitments.
- B: If the colors to the commitments are the same, output `Alice aborted`.

B: If you reached this point, output $g(x)$.

If, at any point, Alice (resp., Bob) sends an incorrectly formatted message or fails to send a message, Alice (resp., Bob) outputs `Bob aborted` (resp., `Alice aborted`). (The number of repetitions $4m \log(1/\varepsilon)$ of the zero-knowledge protocol was chosen in order to make that protocol sufficiently sound.)

Clearly if Alice and Bob are honest, this protocol computes the given functionality. We now sketch why this protocol is a secure implementation of the ideal protocol $(\tilde{A}, \tilde{B})$.

Let us first consider the case of a dishonest Bob B^*. In the zero-knowledge protocol, this dishonest Bob is given input $g(x)$ and his view of the zero-knowledge interaction can be simulated by the corresponding simulator $S_B^*(g(x))$. To simulate B^* in the ideal protocol, $\tilde{B}^*$ first submits his input $\perp$, finds out the value $g(x)$ from the trusted party, then runs the simulator S^* on input $g(x)$ to generate a view for Bob in the real protocol. If at any point S^* simulates an aborting message from Bob's part, $\tilde{B}^*$ scraps the simulation and sends an abort message to the trusted party. Otherwise, $\tilde{B}*$ completes the simulation by producing the output of B^* when evaluated on the simulated view produced by $S_B^*(g(x))$.

Let's now look at the more interesting case of a dishonest Alice A^*. In the first round, A^* can send an arbitrary message r to Bob. The ideal Alice $\tilde{A}^*$, on the other hand, is required to provide an actual input x^* to the trusted party. How is she going to derive x^* based on r?

Here is how we reason about it. Let us look at what (honest) Bob B will do in one run of the zero-knowledge protocol. Depending on the strategy A^*, with some probability p^*, Bob will detect that Alice is cheating in this run and output the message `Alice aborted`. If this p^* is not too small – say larger than $1/2m$, where ε is the hiding parameter of the commitment scheme – then after repeating the protocol for $4m\log(1/\varepsilon)$ times, Bob will detect the abortion with probability $1-\varepsilon$, so if after simulating the interaction between A^* and B for so many rounds, $\tilde{A}^*$ ever detects the message `Alice aborted`, $\tilde{A}^*$ sends an abort message in the ideal protocol and outputs whatever A^* would output. In this case, $(\tilde{A}^*, B^*)$ will be $(\infty, O(\varepsilon))$ indistinguishable from (A^*, B^*) since each of them is $(\infty, O(\varepsilon))$-indistinguishable from the distribution $(A^*, \texttt{Alice aborted})$.

If, on the other hand, p^* is smaller than $1/2m$, it means that with probability $1/2$ over the choice of A^*'s randomness, the probability that Bob detects Alice aborting is less than $1/m$. Since there are m different pairs of commitments that Bob can ask for and none of them lead to abort, the committed values must be of a valid 3-coloring *col* of G (in other words, if at least one edge was improperly colored, Bob would have output `Alice aborted` with probability $1/m$). Then $\tilde{A}^*$ can obtain this coloring *col* by simulating the interaction between A^* and B repeatedly, using B to challenge A^* to obtain the decommitments of the endpoints for every edge of G. Using *col*, which is a valid 3-coloring for G, one can then obtain a string x^* such that $(g(x), x^*) \in R_g$, namely such that $g(x^*) = g(x)$. (The reduction from R_g to $3COL$ we introduced last time also allows us to map proofs for $3COL$ back to proofs for R_g. Once it gets hold of this x^*, $\tilde{A}^*$ can provide it to the trusted party and output the outcome of the simulated interaction between A^* and B as its output.

This is not a proof, but at least I hope it gives some intuition how one could argue that the proposed implementation of the image transmission protocol is secure. To argue security for Bob, we used the following property of our zero-knowledge proof construction: If the prover (in this case, Alice, who could be cheating) convinces Bob to accept with sufficiently high probability, then given Alice's interactive algorithm, it is possible to *extract* a proof (by running this algorithm on various challenge messages). This is called a *knowledge extraction property* of zero-knowledge proofs. Not all zero-knowledge proofs have this property, but fortunately the one we saw last time does.

In order to conform to the convention from last lecture, in this definition we switch the roles of Alice and Bob: Now Bob is the prover and Alice is the verifier.

Definition 2. A (non-interactive) algorithm K is a *knowledge extractor* with threshold η for a zero-knowledge protocol (A, B) for a proof relation R if for every pair $(x, y) \in R$ and every interactive algorithm B^* such that

$$\Pr_{A(x),B^*}[A \text{ accepts}] \geq \eta$$

when given the code of B^* as input, K outputs a proof y^* such that $(x, y^*) \in R$.

We just argued that following algorithm K is a knowledge extractor for B^* with threshold $1-1/2m$ in the zero-knowledge protocol for 3-coloring: First, run B^* to produce candidate commitments $C(v)$ for the colors to all the vertices v. Then challenge B^* on all the edges of the graph to reveal the commitments for all v. If all commitment reveal to valid colors $\{\mathtt{R}, \mathtt{G}, \mathtt{B}\}$ and the revealed coloring *col* is a valid 3-coloring of G, output *col*. Otherwise, repeat the protocol with fresh randomness for B^*, until a valid 3-coloring is obtained.

A zero-knowledge proof that has a knowledge extractor whose running time is ke of the running time of its input program is called a *zero-knowledge proof of knowledge* with *proof extraction overhead* $ke(\cdot)$.

3 Secure two-party computation

We now have all the elements to describe the secure two-party computation protocol for any functionality of the type $(x, y) \to (\perp, g(x, y))$. Our starting point is the protocol (A, B) for this functionality in the honest-but-curious model from Lecture 10. Recall that in this protocol, Alice goes first. We will use $A_1, B_2, A_3, B_4, \ldots$ to denote the algorithms Alice and Bob use to compute the corresponding message in the protocol (given their input, randomness, and messages previously received from the other party).

Secure two-party protocol. Given inputs x for Alice and y for Bob:

1. **(Input commitment phase)** Alice sends a commitment $Com(K_x, x)$ to her input x. The commitment is followed by a zero-knowledge proof of knowledge for $(Com(K_x, x), (K_x, x))$ with respect to the proof relation

$$R_{Com} = \{(y, (K, z)) \colon y = Com(K, z)\}.$$

   Bob sends a commitment $Com(K_y, y)$ to his input y. The commitment is followed by a zero-knowledge proof of knowledge for $(Com(K_y, y), (K_y, y))$ with respect to the proof relation R_{Com}.

2. **(Randomness commitment phase)** Alice chooses a random string r'_A, and sends a commitment $Com(K_A, r'_A)$ followed by a zero-knowledge proof of knowledge for $(Com(K_A, r'_A), (K_A, r'_A))$ with respect to R_{Com}. Bob sends Alice a random string r''_A. Alice sets $r_A = r'_A + r''_A$.

   Bob chooses a random string r'_B, and sends a commitment $Com(K_B, r'_B)$ followed by a zero-knowledge proof of knowledge for $(Com(K_B, r'_B), (K_B, r'_B))$ with respect to R_{Com}. Alice sends Bob a random string r''_B. Alice sets $r_B = r'_B + r''_B$.

3. **(Honest-but-curious protocol emulation phase)** Alice and Bob emulate the honest-but-curious protocol (A, B) when Alice's input is x and her internal randomness is r_A, Bob's input is y and his internal randomness is r_B. The simulation proceeds as follows:

   A: Send the first message $A_1(x, r_A)$ (this is a deterministic function of x and r_A, followed by a zero-knowledge proof of $(A_1(x, r_A), (x, r_A, K_x, K_A))$ for the proof relation R_1:

$(a_1, (x, r_A, K_x, K_A)) \in R_1$ if $a_1 = A_1(x, r_A)$ and the commitment sent by Alice in the input commitment phase equals $Com(K_x, x)$ and the commitment sent by Alice in the randomness commitment phase equals $Com(K_A, r_A + r''_A)$.

B: Upon receiving a_1 with a (valid) zero-knowledge proof, send the message $B_2(y, r_B, a_1)$, followed by a zero-knowledge proof of $(B_2(y, r_B, a_1), (y, r_B, K_y, K_B))$ for the proof relation R_2 defined analogously to R_1.

$\vdots$

B: Upon completion of all the rounds, compute and output B's output on input y, randomness r_B, and exchanged messages $a_1, b_2, \ldots, a_t$.

If at any point Alice sends an invalid message, or Bob does not accept one of her zero-knowledge proofs, Bob outputs `Alice aborted` and halts, and same goes for Bob.

4 General functionalities

So far we only considered secure computations of the type $(x, y) \to (\perp, g(x, y))$, where Bob gets and output and Alice gets nothing. What about general functionalities $(x, y) \to (f(x, y), g(x, y))$?

Before we design a protocol and argue security, we need to define an ideal functionality for this setting. A natural extension would be like this: First, Alice and Bob simultaneously submit their private inputs x and y to the trusted party. The trusted party computes $f(x, y)$ and $g(x, y)$, and sends $f(x, y)$ privately to Alice and $g(x, y)$ privately to Bob. If Alice refuses to submit an input, the trusted party forwards the message `Alice aborted` to Bob and Bob outputs it, and same goes for Bob.

What happens if we want to implement this functionality by a real protocol? It turns out it is not always possible. A natural solution would be to extend the two-party protocol we just described as follows: The commitment phases stay the same. Then we run the emulation phase first for the functionality $(x, y) \to (f(x, y), \perp)$, and then for $(x, y) \to (\perp, g(x, y))$. The problem with this solution is that there is some point in the protocol at which Alice has received her answer $f(x, y)$, but Bob does not yet know his answer $g(x, y)$. If Alice aborts the protocol at this point, her action cannot be simulated in the ideal protocol: There, Alice and Bob find out the values $f(x, y)$ and $g(x, y)$ simultaneously.

This issue has nothing to do with our implementation and is inherent in the problem: Even very simple functionalities like $(x, y) \to (r, r)$, where $r \sim \{0, 1\}$ are not achievable by this definition. No matter how we design the protocol, there will be a point at which one party learns the output and the other one doesn't, and if the first party aborts at this point the outputs of the ideal protocol can never be simulated.

To account for this problem of unfairness, we need to change our model of ideal functionality. There are various possibilities, of which perhaps the simplest is to allow for one of the parties to be unfair, but to require that this unfairness is detected in the protocol implementation.

Ideal two-party protocol for general functionalities: On input x for Alice and y for Bob:

1. Alice sends x privately to the trusted party T. At the same time, Bob sends y privately to T.
2. If Alice did not submit an input, T forwards the message `Alice aborted` to Bob. Bob outputs this message and halts. If Bob did not submit an input, T forwards the message `Bob aborted` to Alice. Alice outputs this message and halts. Otherwise, if x^* and y^* are the inputs submitted by Alice and Bob, T sends the value $f(x^*, y^*)$ privately to Alice.
3. Alice privately outputs the received value from T and sends the message `continue` to T.
4. If T receives the message `continue` from Alice, T sends the value $g(x^*, y^*)$ privately to Bob. Otherwise, T sends the message `Alice is unfair` to Bob.
5. Bob privately outputs the value/message received from T.

A secure implementation with respect to this definition of "ideal two-party protocol" can now be achieved by the protocol suggested above. First, Alice and Bob execute the input commitment phase and randomness commitment phase. Then they jointly emulate the honest-but-curious protocol for $(x, y) \to (f(x, y), \perp)$. If at any point Alice (resp., Bob) sends an invalid message, Alice (resp., Bob) outputs `Bob aborted` (resp., `Alice aborted`). Alice writes the output of this protocol on her private output tape. Then Alice and Bob jointly emulate the honest-but-curious protocol for $(x, y) \to (\perp, g(x, y))$. If at any point Bob sends an invalid message, Alice outputs `Bob aborted`. If at any point Alice sends an invalid message in this phase, Bob outputs `Alice is unfair`. At the end, Bob writes the output of this protocol on his private output tape.