In Lectures 7 and 8 we saw several ingredients that go into the design of secure two-party computation protocols:

- Honest-but-curious two-party computation: Alice has input x, Bob has input y, Alice learns nothing, Bob learns $f(x, y)$ but nothing else.
- Commitments: Sender can commit to any value x of his choice in a hiding (Receiver gets no information) and binding (Sender cannot decommit to any value other than x) manner.
- Universal zero-knowledge proofs: Prover can convince verifier that any statement x (for which he knows a proof π) is true without revealing anything beyond its veracity.

In this lecture we apply them to obtain two-party computation that is secure against malicious parties that might not play by the rules.

1 Defining secure two-party computation

A protocol between Alice and Bob for computing $f(x, y)$ should be called secure if the views of a cheating Alice interacting with Bob and a cheating Bob interacting with Alice can be simulated. But what should the simulators be given as their inputs? Let's see what happens when Alice's simulator is given x and Bob's simulator is given y and $f(x, y)$ like in the honest-but-curious case.

Consider the case of the AND function and assume Bob's true input y is 0. A cheating Bob can always pretend that his input is 1 and learn Alice's input x. His view cannot be simulated from y and $f(x, y) = 0$. This "attack" is unavoidable and should not really count as cheating as there is no way to distinguish Bob's true input y from the input y^* that he provides to the protocol.

Why do Alice and Bob want to run a protocol in the first place? The purpose of two-party computation is to replace a trusted authority that performs the computation privately for them. But Alice and Bob an submit false inputs even to a trusted authority. On a dating website, Bob can falsely claim that he is interested in Alice to find her preference. In Yao's millionaire problem, Bob can falsely claim that he is a billionaire to win the contest.

In the presence of a trusted authority, the *ideal secure two-party functionality* is computed like this.

1. Alice and Bob simultaneously submit inputs x and y, respectively, to the trusted party.
2. The trusted party computes $f(x, y)$ and sends the answer to Bob (unless one party failed to provide an input in which case it declares failure).

In the ideal functionality a party can always submit a false input. This is unavoidable in any implementation. The security requirement should say that this is the only way to "cheat". One elegant way to specify this is to say that whatever a cheating party can learn by running the protocol can be simulated by the same cheating party in the ideal functionality.

Definition 1. A two-party protocol is (s, ε)-*secure for Alice* if for every circuit B^* there is a simulator $\tilde{B}^*$ (of size at most oh larger) such that B^*'s view when interacting with Alice in the protocol is (s, ε)-indistinguishable from $\tilde{B}^*$'s output when interacting with the trusted party in the ideal functionality.

For example, if Bob tries to cheat by pretending his input is y^*, his behavior can be simulated by a $\tilde{B}^*$ that submits y^* to the trusted party, learns $f(x, y^*)$ and then simulates cheating Bob's view from this information.

Now let's consider security for Bob. Since Alice receives no output in the ideal functionality, by analogy we may try to say security for Bob simply means that Alice's view in the protocol can be simulated without extra information. This sounds like a strong guarantee, but here is an example in which what Bob gets from the protocol is not the same as what happens in the ideal functionality.

Consider the case when f is a function of x only and does not depend on Bob's input y. A protocol for such function should perform *image transmission*: Bob should learn $f(x)$ but nothing beyond that. In the honest-but-curious security model there is a particularly simple image transmission protocol: Alice sends $f(x)$ to Bob. Under the requirement of Definition 1, this protocol would remain "secure" for Bob because Alice receives no messages so her view is trivially simulatable.

Now think of a cheating Alice who samples some h^* that is not even a possible image of f and sends it to Bob (who accepts). In the ideal functionality, this type of interaction cannot occur: Bob's output there is always of the form $f(x^*)$ for some x^*. To prevent this type of attack, the full definition of security requires that Alice's view can be simulated even when conditioned on Bob's output (see Figure 1).

Definition 2. A two-party protocol is (s, ε)-*secure for Bob* if for every circuit A^* there is a simulator $\tilde{A}^*$ (of size at most oh larger) such that the random variable consisting of A^*'s view and Bob's output in the protocol is (s, ε)-indistinguishable from $\tilde{A}^*$'s output together with Bob's output in the ideal functionality.

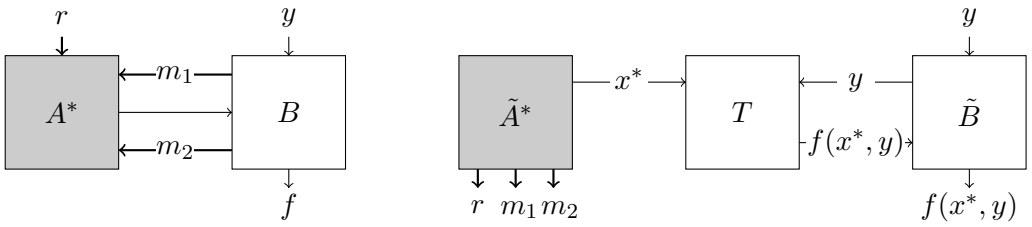

Figure 1: Security of two-party computation for Bob: (a) the protocol execution; (b) the ideal functionality. Definition 2 requires that for every A^* there is a $\tilde{A}^*$ for which the random variables (r, m_1, m_2, f) are indistinguishable in the two executions.

Under Definition 2 this image transmission protocol is no longer secure. The security definition requires that this cheating Alice A^*, which does nothing but sample some h^*, can be simulated by a cheating Alice $\tilde{A}^*$ in the ideal functionality so that her output is indistinguishable from Bob's output h^* in the protocol. This cheating Alice $\tilde{A}^*$ must therefore use h^* to sample some input x^* so that Charlie's output $f(x^*)$ is indistinguishable from h^*. Such an x^* might not even exist: For example, if $f\colon \{0,1\} \to \{0,1,2\}$ is the identity function $f(x) = x$ then the cheating Alice that sends 2 to Bob (who then outputs it) cannot be simulated by any cheating Alice in the ideal functionality under Definition 2.

One way to guarantee the existence of x^* is to augment the simple protocol with a zero-knowledge proof that h is in the range of f. This is a step in the right direction, but still poses a challenge in the simulation. Suppose that $f\colon \mathbb{Z}_q \to \mathbb{G}$ is the exponentiation function $f(x) = g^x$ for a generator g of $\mathbb{G}$. This f is a bijection, so the statement "h^* is in the range of f" is vacuously true. Alice's simulator $\tilde{A}^*$, however, still has to produce an input x^* such that $f(x^*)$ looks exactly like cheating Alice's output h^*.

2 Proofs of knowledge

A proof of fact is a protocol by which Prover convinces Verifier that a given statement is a fact, i.e. it has a proof π. In a proof of knowledge, Prover needs to convince Verifier not only that the statement is true, but that he knows the proof π. Schnorr's protocol is an example of a proof of knowledge for discrete logarithms: The statement is some h in $\mathbb{G}$ and the proof if the (unique secret key) x for which $g^x = h$ (the proof relation consists of the pairs (g^x, x) as x ranges over $\mathbb{Z}_q$). The existence of x is clear; what is called into question is Prover's knowledge of x.

In the case of Schnorr's protocol, the proof of security showed that if a cheating prover passes validation then we can use this prover to calculate the discrete logarithm of h. In the general setting, from any prover that passes verification of a statement it should be possible to recover a proof of that statement. The program that performs the proof recovery is called a knowledge extractor. It size will in general depend on the size of the cheating prover.

Definition 3. A Prover-Verifier protocol for a given proof relation is a *proof of knowledge* if for every prover P^* there exists a *knowledge extractor* circuit K of size oh larger such that if P^* passes verification (with probability greater than a given threshold) on input x, then $K(x)$ outputs a proof π for x with probability at least $1/2$.[1]

Any (complete) proof of knowledge is a proof of fact because the knowledge extractor cannot possibly extract a "proof" of a false statement. The converse is not true in general as illustrated by the example of discrete logarithms. However, the GMW protocol happens to also be a proof of knowledge. Since the 3-coloring proof relation is complete, this protocol can be used to provide zero-knowledge proofs of knowledge for any statement.

Theorem 4. *The GMW protocol is a proof of knowledge with extraction overhead $oh(t) = mt + O(m)$, assuming P^* passes verification with probability at least $1 - 1/2m$ and the underlying commitments are perfectly binding.*

If the protocol is repeated r times, the verification probability threshold drops to $(1 - 1/2m)^r$, so it can in principle be made negligibly small.

Proof. The knowledge extractor runs P^* to produce candidate commitments $C_1^*, \ldots, C_n^*$ to the colors of all the graph's vertices. Then it challenges the cheating prover to decommit C_v^* and C_w^* for every edge (v, w) of the graph. If in all the challenges every C_v^* decommits to the same color $\pi_v \in \{\mathtt{R}, \mathtt{G}, \mathtt{B}\}$, the extractor outputs π as the coloring.[2]

If P^* passes verification with probability at least $1 - 1/2m$ then with probability at least half each of these m decommitments reveals a pair of different colors. Since the binding is perfect, the decommitments to C_v^* for any given v in different challenges must be consistent. Then π is a valid 3-coloring of G. □

The proof of Theorem 4 has a similar structure to the security proof for Schnorr's protocol. In both cases the knowledge extractor simulates the cheating prover on the same commitment then reconstructs the proof from its responses to different challenges.

[1] The extraction probability can in fact be amplified all the way to 1 by a small price in the extraction overhead.

[2] This knowledge extractor does not find out the colors of isolated vertices but these are irrelevant.

3 The Goldreich-Micali-Wigderson compiler

Armed with zero-knowledge proofs of knowledge, consider the following image transmission protocol: Alice sends $h = f(x)$ to Bob together with a zero-knowledge proof of knowledge (with respect to the proof relation $(f(x), x)$). Bob declares failure if Alice's proof doesn't pass verification.

This is certainly secure for Alice, since Bob's view $f(x)$ can be simulated from the same output in the ideal functionality.[3] Now consider a Alice who sends some h^* as her message. If h^* passes verification, then the knowledge extractor can extract a preimage x^* for h^* under f. If Alice's simulator $\tilde{A}^*$ submits this x^* as her input in the ideal functionality, the output $h(x^*)$ is indistinguishable from h^* so the protocol is also secure for Bob.

This idea can be extended to handle any *deterministic* two-party computation protocol (A, B) that is secure against honest-but-curious parties. Assuming Alice goes first, in any such protocol Alice's first message m_1 is a function $A_1(x)$ of her input x, Bob's first message m_1' is a function $B_1(y, m_1)$ of his input y and Alice's message m_1, and so on. After r rounds Bob outputs $B_r(y, m_1, \ldots, m_r)$.

The GMW compiler. Given inputs x for Alice and y for Bob:

Input commitment phase: Alice sends a commitment $C_x = Com(x)$ to her input x, followed by a zero-knowledge proof that she knows x that commits to C_x. Bob does the same for his input y. Each party verifies the proof of knowledge.

Protocol emulation phase: Alice and Bob emulate the honest-but-curious protocol (A, B). Alice's first message $m_1 = A_1(x)$ is followed by a zero-knowledge proof that there exists an x such that C_x is a commitment to x and $m_1 = A_1(x)$. Bob verifies Alice's proof and sends his first message $m_1' = B_1(y, m_1)$ is followed by a zero-knowledge proof that there exists a y such that C_y is a commitment to y and $m_1' = B_1(y, m_1)$, and so on. After r rounds Bob outputs $B_r(y, m_1, \ldots, m_r)$. (If any proof doesn't verify the party declares failure.)

The proof of security is not difficult, but perhaps a bit tedious, so we just explain the role of the different ingredients. Let's look at security for Bob. The commitments ensure that Alice's input x^* is consistent throughout the protocol. The proofs of knowledge in the input commitment phase ensure that Alice's simulator $\tilde{A}^*$ can reconstruct the input x^* for the ideal functionality. The proofs of fact in the protocol emulation phase ensure that Alice's messages are consistent with what they would be an honest-but-curious interaction assuming her input was x^*. The zero-knowledge of all the proofs and the assumed honest-but-curious security of the original protocol ensure that Alice learns no information.

To obtain a secure two-party protocol for arbitrary functionalities we would like to apply the GMW compiler to Yao's protocol. One complication is that Yao's protocol is randomized. Randomness played a crucial role in the underlying AND protocol that we used to implement oblivious transfer. The security of that protocol relied on the idealistic assumption that Bob properly samples public and/or private keys. Without a proof that the sampling was performed correctly, which in particular means that Bob used a reliable source of randomness, all security bets are off.

The last ingredient is a *private coin sampling protocol* by which Bob can output a bit which is guaranteed to be random to an honest Alice, but about which Alice doesn't learn anything. This can be reduced to a secure implementation of the ideal XOR function $x \oplus y$: If the honest party chooses a random input, the output is random and independent of the one provided by the cheating party.

[3] Bob's view also includes his verifier's view in the zero-knowledge interaction, but this part is simulatable by the zero-knowledge property.

XOR protocol.

1. Bob sends a commitment $C = Com(y)$ to his input y.
2. Alice sends x.
3. Bob outputs $r_B = x + y$.

Claim 5. *If commitments are (s, ε)-secure then the XOR protocol is $(s-t, \varepsilon)$-secure for Bob against cheating Alices of size at most t.*

Proof. Alice's view consists of a commitment $Com(y)$ to Bob's input. Upon seeing it, she generates a response $x^*(Com(y))$ to Bob. The simulator $\tilde{A}^*$ outputs a simulated commitment Sim as her view and provides $x^*(Sim)$ as her input in the ideal functionality. The security requirement for Alice is that $(Com(y), x^*(Com(y)) + y)$ is indistinguishable from $(Sim, x^*(Sim) + y)$ for all y. If these can be distinguished for some y by some D, then $D(C, x^*(C) + y)$ would distinguish real and simulated commitments with the same advantage. (The view of a cheating B^* consists of a random bit, so it can be simulated by a $\tilde{B}^*$ that submits an arbitrary input and outputs an indepednent random bit.) □

To handle randomized protocols, the GMW compiler is augmented with a randomness commitment phase, following the input commitment phase, in which Alice and Bob run the XOR protocol to generate private randomness for Bob (and for Alice if needed by reversing their roles). Claim 5 guarantees that a cheating Alice does not obtain information about Bob's private randomness r_B. Alice also needs to be guaranteed that Bob followed the rules of the XOR protocol in generating his randomness. To do this, he accompanies any of his messages (which may depend on r_B) with a zero-knowledge proof that r_B is the output of the XOR protocol, i.e. that there exists a y such that C is a commitment of y and $r_B = x + y$.

4 Fairness

So far our discussion has been restricted to asymmetric two-party computations. What about symmetric computations in which both Alice and Bob should obtain the output $f(x, y)$?

For honest-but-curious parties there is no real difference between symmetric and asymmetric computations: A protocol for symmetric computation can be obtained from two runs of the asymmetric protocol with the roles of Alice and Bob reversed the second time. If the parties are malicious, however, this strategy is problematic because if say Bob finds $f(x, y)$ first (before Alice has learned anything) he may halt and refuse to send any more messages. This wouldn't be fair to Alice.

A *fair two-party computation* is a secure implementation of the following ideal functionality:

1. Alice and Bob simultaneously submit inputs x and y, respectively, to the trusted party.
2. The trusted party computes $f(x, y)$ and sends the answer simultaneously to Bob and Alice (unless one party failed to provide an input in which case it declares failure).

In particular, a fair two-party computation of the XOR function in which both Alice and Bob submit random inputs is a mechanism for Alice and Bob to obtain the same random bit r that is sampled by trusted Charlie. This is the *fair coin toss specification*. It turns out that a secure fair coin toss is impossible to implement efficiently! To get a sense of the difficulty, consider the following variant of the XOR-based coin tossing protocol.

1. Bob chooses a random bit y an sends a commitment $C = Com(y)$ to Alice.
2. Alice sends a random bit x to Bob.
3. Bob computes $r = x + y$, forwards it to Alice, and outputs it.
4. Alice outputs the bit forwarded to her by Bob.

If both parties follow the protocol, their common output is clearly a random bit. Bob, however, finds out the value of this random bit first. If it is not to his liking — say he wants a zero but r is equal to one — he can refuse to forward r to Alice. In this case, Alice still needs to produce an output which is indistinguishable from her output in the fair coin toss specification, namely a random coin toss. What should she do?

One option is that if Bob aborts in Step 3 then Alice samples outputs a random bit a on her own. But this protocol is still unfair to Alice: Alice's output is a random bit a conditioned on $r = 1$, and r itself conditioned on $r = 0$, which makes her bit heavily biased towards 0. Can Alice perhaps sample a in a biased manner to mitigate Bob's possible abortion?

Cleve showed that in any candidate k-message coin tossing protocol, one of the parties can introduce a bias of at least $1/(8k + 1)$ in the output, even if its only available strategies are to follow the protocol or abort at some point. Let me try to give some intuition about why fair coin tossing is impossible in two rounds. Suppose that Alice sends a message, Bob replies, and then they agree on a fair coin toss r. If Alice aborts Bob chooses the output b on his own. If Bob fails to reply Alice chooses the output a on her own.

For the protocol to be fair, the marginal distributions of a, b, and r should all be very close to uniform. Before his decision to abort, Bob knows the protocol outcome r. Unless a and r are strongly correlated (i.e. a is almost determined by r), Bob can induce some bias by strategically choosing to abort or release r just like in the example we saw. But if a and r are strongly correlated, then Alice knows the outcome r even before she sends any messages. So Alice can also induce bias by a strategic decision, unless b and r also happen to be strongly correlated.

The only remaining possibility is that both a and b are strongly correlated with r. This means a and b are individually random and almost always equal or almost always different. But then Alice and Bob know a common random bit before any interaction between them has taken place, which is clearly impossible![4]

In conclusion, fair secure two-party computation is impossible to achieve for all but a handful of functions f.[5] The next best thing is the following type of ideal functionality.

1. Alice and Bob simultaneously submit inputs x and y, respectively, to the trusted party.
2. The trusted party reveals $f(x, y)$ to Bob. Bob can then choose to continue or abort.
3. If Bob aborts the trusted party announces this. Otherwise it reveals $f(x, y)$ to Alice.

The natural extension of the GMW protocol to the symmetric setting satisfies this specification. Although this is unfair to Alice, it has the advantage that Bob's unfairness can be detected. In some infrastructures like blockchains with "smart contracts" Bob's fairness can be incentivized by introducing penalties.

[4] For protocols with more messages, this argument essentially eliminates the last round of interaction and can be applied inductively to rule out fairness.

[5] A notable exception is the AND function.

5 Secure multiparty computation

In a secure multiparty computation, several parties want to perform a joint computation on their private inputs without revealing any information. Let's consider the case of three parties Alice, Bob, and Charlie. Even in the honest-but-curious model there are two possible definitions of security:

1. No party finds out any information beyond its output.
2. In addition, no two parties find out any information beyond their outputs.

It is possible to obtain the stronger security requirement by extending the two-party techniques that we just discussed. Instead we describe the perfectly secure protocol of Ben-Or, Goldwasser, and Wigderson (independently discovered by Chaum, Crepeau, and Damgård) that satisfies the weaker one. This protocol assumes that private communication channels between all pairs of parties are available.

The functions to be computed are represented by *arithmetic circuits*. Inputs and outputs belong to some finite field $\mathbb{F}$ and the gates of the circuit compute additions (more generally, linear combinations) and multiplications. Boolean circuits with AND and XOR gates can be viewed as arithmetic circuits over the binary field $\mathbb{F}_2$.

Let's first discuss the special but important case in which Charlie wants to privately calculate some linear function of Alice's input a and Bob's input b , e.g. $f(a, b) = 2a + 3b$. To do this, Alice and Bob first jointly sample a random element $r \sim \mathbb{F}$. Then Alice sends Charlie the value $2a + r$, Bob sends Charlie $3b - r$, and Charlie outputs the sum of the two messages. Alice's and Bob's views consist of a single random bit so the protocol is secure against them, while Charlie observes two random numbers that add up to his output. This is a view that he can simulate from his output.

The BGW protocol consists of three phases. In the setup phase, each party's input(s) is distributed to the other parties via a suitable secret sharing scheme. In the computation phase, parties compute shares representing the values at all the wires in the order of circuit evaluation. In the reconstruction phase, the shares representing the output wire(s) are combined to obtain the desired output(s).

The underlying secret sharing scheme will be Shamir's scheme from Lecture 1 (but other choices are also possible). Alice's, Bob's, and Charlie's shares are the values $\ell(1)$, $\ell(2)$, and $\ell(3)$, where ℓ is a random linear function conditioned on the secret being $\ell(0)$, i.e. the function $\ell(t) = \text{secret} + rt$ for a random r. In this scheme no party has any information about the secret but any two can reconstruct it.

In the setup phase each party shares their input via this scheme. For example Alice sends the values $a + r$, $a + 2r$, and $a + 3r$ to herself, Bob, and Charlie, respectively. In the reconstruction phase, the output(s) are interpolated from their share(s). For example, Alice can uniquely determine her output from hers and Bob's shares of it.

It remains to describe the computation phase. Suppose Alice, Bob, and Charlie want to evaluate the shares representing the output of a plus gate $p(0) + q(0)$. Each knows their share $p(t)$ of $p(0)$ and $q(t)$ of $q(0)$. Each declares $p(t) + q(t)$ to be their share of the output $p(0) + q(0)$. Since p and q are linear functions, so is $x + y$ and each party ends up with a valid share for the output. This step doesn't involve communication so it preserves security.

In the case of a times gate $p(0) \cdot q(0)$, each party multiplies its shares $p(t)$ and $q(t)$. If p and q are linear functions describing the parties shares, then $s = pq$ is a quadratic function whose value at zero equals the gate output $p(0) \cdot q(0)$. The values $s(1)$, $s(2)$, and $s(3)$ still uniquely specify the

desired output $r(0)$. The relation is given by the Lagrange interpolation formula

$$s(0) = 3s(1) - 3s(2) + s(3).$$

To carry on the execution the parties need to securely re-share the value $s(0)$ via a new linear function ℓ. The value $\ell(0)$ should equal the output gate value $s(0)$. Alice's value $\ell(1)$ should be a random number $r \sim \mathbb{F}$. Bob's and Charlie's shares $\ell(2)$ and $\ell(3)$ are then determined by interpolation:

$$\ell(t) = (1-t)s(0) + tr = (3(1-t)s(1) + tr) - 3(1-t)s(2) + (1-t)s(3).$$

To calculate $\ell(2)$ and $\ell(3)$, Bob and Charlie need to learn the value of some linear function of the other two parties' secret information and nothing else, which we showed how to do.

The protocol easily extends to more than three parties. It is secure against any coalitions of fewer than $n/2$ honest-but-curious parties. A variant of it is secure and fair against fewer than $n/3$ malicious parties. This is optimal for statistically secure protocols. In contrast, protocols based on oblivious transfer provide security against any number of honest-but-curious parties or at most $n/2$ malicious parties.