CS 120/ E-177: Introduction to Cryptography

Salil Vadhan and Alon Rosen Dec. 12, 2006

Lecture Notes 19 (expanded): Secure Two-Party Computation

Recommended Reading.

- Goldreich Volume II 7.2.2, 7.3.2, 7.3.3.

1 A Tale of Two Lovers

Alice and Bob would like to determine whether they love each other. However, Alice is afraid to reveal her love to Bob only to find out that he actually doesn't love her. Bob is afraid of the very same thing. They decide to run a cryptographic protocol.

Alice holds an input bit $a \in \{0,1\}$ representing whether she loves Bob ($a = 1$) or not ($a = 0$). Bob holds an input bit b representing whether he loves Alice or not. Their goal is to somehow determine whether both $a = 1$ and $b = 1$. In other words, they wish to jointly compute the function

$$f(a,b) = a \wedge b.$$

The ideal thing for them would be a protocol that reveals the value of $f(a,b)$ to both of them, but does not reveal any more information on their respective inputs than what is “necessary." Clearly, having Alice and Bob send each other their inputs on the clear is not a good idea (what if $f(a,b) = a \oplus b$?). However, it is not at all clear how to do this in a different way.

In this lecture, we will see that, somewhat surprisingly, cryptography enables a solution to Alice and Bob's problem. In fact, it will yield a solution to a more general (and challenging!) problem. Under appropriate complexity assumptions, it is possible to securely compute *any* efficiently computable two-party function $f(a,b)$ (where a, b are not necessarily bits).

2 Defining Security

Before they even attempt to design a solution, Alice and Bob have to define what it means for a protocol to be “secure." So they decide to start by listing security properties that they would like the protocol to satisfy:

Correctness. If both Alice and Bob follow the protocol's instructions, then their "local" outputs should equal $f(a,b)$.

Privacy. The protocol reveals no information to Alice beyond what is revealed by $f(a,b)$ (same for Bob), For example if $f(a,b) = a \wedge b$ and $a = 0$ then $f(a,b) = 0$ regardless of the value of b, and the protocol's execution should not enable Alice to tell whether $b = 0$ or $b = 1$.

Input independence. Alice's input to the protocol should not depend on Bob's input (same for Bob). This is closely related to Privacy. For example, Alice should not be able to set her input a to be equal to b. This is because in such a case Alice will be able to look at the output of the protocol (e.g. $f(a,b) = a \wedge b$) and learn the value of Bob's input (why?).

These properties are very desirable in many protocol settings and make sense also if more than two parties are participating. For example, in an "electronic voting" protocol privacy should guarantee that, modulo the outcome of the election, my vote is not revealed.

Another example would be a protocol for contract bidding where the inputs to the protocol correspond to, say the bids of the individual bidders. In this case, not only privacy might be important but also input independence seems to be a crucial property.

2.1 Semi-Honest Adversaries

In this lecture we will focus on *semi-honest* adversaries. These are adversaries that faithfully follow the protocol's prescribed instructions, but nevertheless try to learn something from the protocol's execution (by keeping a record of all their intermediate computations).

Even though it sounds rather weak, the semi-honest model is of great interest by itself and designing protocols that withstand attacks of a semi-honest adversary is already a non-trivial task. (Notice that having the parties send their inputs on the clear would not yield a secure protocol even against semi-honest adversaries.)

The treatment of the semi-honest case will be justified in the next lecture, where we will see how to transform any protocol that is secure against semi-honest adversaries into a protocol that withstands *arbitrary* malicious deviations from the protocol's instructions.

2.2 Security Against Semi-Honest Adversaries

We consider a two party protocol (A, B). The protocol consists of two parties that exchange messages in alternating turns. The messages are determined by the parties' (private) inputs and coin-tosses, as well as on the previous messages that they have received. We will say that a protocol (A, B) computes a function $f(a, b)$ if whenever both A and B follow the protocol's instructions, on inputs a and b respectively, then their local outputs equal $f(a, b)$

A party's *view* of a protocol's execution consists of its input, coin-tosses, and received messages. We let $\mathsf{VIEW}_A^{(A,B)}(a, b, 1^n)$ denote a random variable that is distributed according to A's view when the parties' inputs are a, b respectively, the security parameter is n, and the coin tosses of both parties are chosen uniformly at random. B's view is symmetrically defined.

One could think of the view of a party as the "information" that this party is obtaining from the protocol's execution. A semi-honest adversary may apply an arbitrary polynomial time program to the view (this corresponds to the distinguisher).

Our definition of security will require that whatever can be (efficiently) obtained from a party's view could be essentially obtained from the input and output available to that party. This is formalized by exhibiting the existence of a probabilistic polynomial time machine (*simulator*) that, given only the input and output available to the party, is able to produce views that are essentially identical to that party's views in actual protocol executions.

Definition 1 *Let $f(a, b)$ be a (deterministic) two-input function. A two-party protocol (A, B) for computing f is said to be* secure *if there exists PPT algorithms ("simulators") S_A and S_B such that for any a, b*

$$\left\{S_A(a, f(a,b), 1^n)\right\}_{n \in N} \overset{c}{\equiv} \left\{\mathsf{VIEW}_A^{(A,B)}(a, b, 1^n)\right\}_{n \in N}$$

$$\left\{S_B(b, f(a,b), 1^n)\right\}_{n \in N} \overset{c}{\equiv} \left\{\mathsf{VIEW}_B^{(A,B)}(a, b, 1^n)\right\}_{n \in N}$$

where $\overset{c}{\equiv}$ denotes computational indistinguishability.

Notice the similarity to semantically secure encryption. There, we required that whatever is computable about the plaintext with the ciphertext is actually computable without the ciphertext.

2.3 Comments

In the semi honest case input independence is trivially guaranteed. However, in the case of malicious adversaries this is indeed a concern. The reason for this is that a malicious adversary may modify its own input as a function of the messages it sees. There are many issues/variants that are not covered here:

- Randomized functions,
- parties could have different outputs (i.e. $f_1(a,b)$ and $f_2(a,b)$).
- malicious adversaries,
- static/dynamic/adaptive/non-adaptive corruption,
- fairness.

3 Computing the AND of Two Bits

Back to Alice's and Bob's problem. We would like to design a protocol that computes the function $f(a,b) = a \wedge b$ where $a, b \in \{0,1\}$. The central tool that we will use will be 1-out-2 oblivious transfer.

3.1 Oblivious Transfer

Roughly speaking, *1-out-2 Oblivious Transfer* (OT_1^2) is a protocol between Alice (the *sender*) that holds two values (say bits) s_0, s_1 and Bob (the *receiver*) that holds a "choice" bit $c \in \{0,1\}$. At the end of the protocol, Bob outputs the value of s_c and should learn nothing else. Alice learns nothing. In the language of our definition of security this corresponds to a protocol for securely computing the two-input function $f(a,b) = (f_1(a,b), f_2(a,b))$ defined as

$$f((s_0, s_1), c) = (\lambda, s_c)$$

where λ denotes the empty string and corresponds to Alice's empty output. We next present a construction that is secure for semi-honest adversaries. Let $\mathcal{F} : \{f_k : D_k \to D_k\}_{k \in K}$ be a family of trapdoor permutations and let $\{b_k : D_i \to \{0,1\}\}_{k \in K}$ be a collection of trapdoor predicates for F. Consider the following protocol for OT_1^2:

Common input: Security parameter $n \in N$.

Input to Alice: $s_0, s_1 \in \{0,1\}$.

Input to Bob: Choice bit $c \in \{0,1\}$

$A \to B$: Pick $(k,t) \overset{R}{\leftarrow} G(1^n)$ for the trapdoor permutation and send k to Bob.

$B \to A$: Pick $x, y \overset{R}{\leftarrow} D_k$. Set $y_c = f_k(x)$ and $y_{1-c} = y$. Send y_0, y_1 to Alice.

$A \to B$: Use t to invert y_0, y_1 under f_i. Let x_0, x_1 be the corresponding preimages. Use x_0 to "encrypt" s_0 and x_1 to "encrypt" s_1. This is done by setting $z_0 = b(x_0) \oplus s_0$ and $z_1 = b(x_1) \oplus s_1$. Send z_0, z_1 to Bob.

B: Use $x = x_c = f_k^{-1}(y_c)$ to compute $s_c = b_k(x_c) \oplus z_c$.

The above protocol can be also generalized to work for OT_1^ℓ for any small (e.g. constant) $\ell \in N$ (i.e., where Alice holds k values $s_1, \dots, s_\ell$ and Bob's choice is an index $i \in [\ell]$).

Proposition 2 *Suppose that F is a family of trapdoor permutations. Then, (A, B) is a OT_1^2 protocol against semi-honest adversaries.*

Proof Sketch: Consider a machine $S_A((s_0, s_1), \lambda, 1^n)$ that operates as follows:

1. Pick $(k, t) \overset{R}{\leftarrow} G(1^n)$ for the trapdoor permutation using coin tosses r.
2. Pick two uniformly chosen strings $y_0, y_1 \in D_k$.
3. Output $(s_0, s_1), r, (y_0, y_1)$.

Note that the strings y_0, y_1 that appear in S_A's output are *identically* distributed to strings y_0, y_1 that are sent by Bob in an actual execution of the protocol (A, B). This is because when Bob is applying f_k to a uniformly chosen $x \in D_k$ the result is a uniformly chosen $y \in D_k$. Next, consider a machine $S_B(c, s, 1^n)$ that acts as follows:

1. Pick $(k, t) \overset{R}{\leftarrow} G(1^n)$ for the trapdoor permutation.
2. Pick $x, y \overset{R}{\leftarrow} D_k$ using coin tosses r. Set $y_c = f_k(x)$ and $y_{1-c} = y$.
3. Compute $z_c = b_k(x) \oplus s$ and randomly choose z_{1-c} .
4. Output $c, r, (k, (z_0, z_1))$.

Note that except (z_0, z_1) the output of S_B is distributed identically to the corresponding part of Bob's view. This holds even if we include z_c (which equals $b_k(x) \oplus s$ both in the actual execution and in the execution of S_B). Thus, the only difference is in the value of z_{1-c}. The latter's indistinguishability from the corresponding value in actual executions of the protocol follows from the fact that the function $b_k()$ is a hard-core predicate for the family F. □

3.2 Using OT for Computing AND

Given a protocol for OT_1^2, the task of computing the function $f(a, b) = a \wedge b$ becomes simple. Suppose Alice has input a and Bob has input b. To compute $f(a, b) = a \wedge b$ we let them run the OT_1^2 protocol with Alice as the sender and Bob as the receiver in the following way:

A: Set $(s_0, s_1) = (0, a)$.

B: Set $c = b$.

$A \leftrightarrow B$: Run the OT_1^2 protocol with $((s_0, s_1), c)$ as input. Bob gets the value of s_c as output.

B: send s_c to Alice.

A, B: output s_c as the result of the protocol for computing $f(a, b)$.

It can be seen that $s_c = 1$ if and only if $a = b = 1$. Thus, the protocol indeed computes $f(a, b) = a \wedge b$.

Claim 3 *Suppose that the OT is secure. Then, protocol (A, B) securely computes $f(a, b) = a \wedge b$.*

4 Computing an Arbitrary Function

We next sketch the construction of a secure two-party protocol for computing an arbitrary $f(a, b)$.

4.1 Computing Functions with Boolean Circuits

A Boolean Circuit is a DAG whose edges (*wires*) are labeled with a value in $\{0, 1\}$, and whose nodes (*gates*) correspond to Boolean operations (e.g. AND, OR, NOT). We may assume without loss of generality that nodes in the circuit have maximal in-degree 2.

The computation takes place in the natural way, where for each gate we label its outgoing wire(s) with the value that corresponds to the output of the gate function when applied to the $\{0, 1\}$ values with which the two incoming wires are labeled. For example if the gate corresponds to the AND function and its incoming wires are labeled with a_i, b_i respectively then its outgoing wire(s) will be labeled with the value of $a_i \wedge b_i$.

The inputs a, b of the function correspond to the value of *input wires* (which are "sources" in the DAG). The inputs a, b are viewed as bit strings and each wire corresponds to one bit of input. The output of the function corresponds to certain *output wires* (which are "sinks" in the DAG) and equals the value these wires are labeled with.

Fact 4 *Any polynomial time computable function $f(a, b)$ can be computed by a polynomial size Boolean Circuit with only AND and NOT gates.*

4.2 Computing with Shared Inputs

Alice and Bob hold inputs $a^1, \ldots, a^n$ and $b^1, \ldots, b^n$ respectively, where a^i, b^i are bits. They wish to jointly compute the function $f((a^1, \ldots, a^n), (b^1, \ldots, b^n))$. To do so, they start by exchanging "shares" of each of their input bits a^i and b^i. Thus is done in the following way. Suppose that Alice would like to share an input bit a and Bob would like to share an input bit b:

$A \to B$: Pick $a_1 \overset{R}{\leftarrow} \{0, 1\}$ and set $a_2 = a \oplus a_1$. Send a_2 to Bob.

$B \to A$: Pick $b_2 \overset{R}{\leftarrow} \{0, 1\}$ and set $b_1 = b \oplus b_2$. Send b_1 to Alice.

Note that $a_1 \oplus a_2 = a$ and $b_1 \oplus b_2 = b$ so a_1, a_2 can be thought of "shares" of a and b_1, b_2 can be though of "shares" of b. Also note that following the above process we have:

1. Alice holds "shares" a_1, b_1 of a and b respectively.
2. Bob holds "shares" a_2, b_2 of a and b respectively.
3. Neither a_1 or a_2 individually reveal any information about a.
4. Neither b_1 or b_2 individually reveal any information about b.

Alice and Bob repeat this process for every one of their input bits a^i and b^i. As a consequence, they both hold shares for all of the circuit's input wires.

Their objective is, given shares of the form a_1, b_1 and a_2, b_2 for two wires that enter a certain gate, to jointly compute shares for the label of the corresponding output wire(s). If they can do this for any given gate, they can eventually go through all the gates of the circuit (one by one) while computing new shares for all "intermediate" wires.

Eventually they reach the output wires of the circuit. If all goes as planned, they are holding shares for the output wires that encode the value of $f((a^1, \ldots, a^n), (b^1, \ldots, b^n))$. Alice and Bob can then send each other their shares so that they can both reconstruct the output of the function.

The key for implementing the above idea in a secure manner is to insure that the joint computation of shares of intermediate wires: (1) results in correct shares for the outgoing wire, and (2) does not reveal anything about the values shared (both the inputs to the gate and the outputs).

4.3 Computing NOT with Shared Inputs

Suppose Alice and Bob hold shares a_1 and a_2 for a bit a. To jointly compute the function $f(a) = \neg a$, we let Alice simply negate her share a_1. This results in new shares $c_1 = \neg a_1$ and $c_2 = a_2$, whose combination $c = c_1 \oplus c_2$ equals $\neg a$.

4.4 Computing AND with Shared Inputs

The more involved case involves the joint computation of the AND function. Suppose Alice and Bob hold shares a_1, b_1 and a_2, b_2 for bits a and b respectively. Their objective is to jointly compute uniformly chosen shares c_1 and c_2 for the value $c = a \wedge b$. Viewing the AND function as the multiplication of two bits modulo 2 and the XOR function as addition modulo 2, we can restate the problem as wanting to provide each party with a *random* share of the value

$$(a_1 + a_2) \cdot (b_1 + b_2).$$

To this end, we let Alice pick a random bit c_1 and engage in a 1-out-4 OT (OT_1^4) protocol with Bob, where Alice's values are defined as follows (values in rightmost column are taken modulo 2):

Index of the secret	B's shares (a_2, b_2)	Value of the secret (B's output)
1	$(0,0)$	$c_1 + a_1 b_1$
2	$(0,1)$	$c_1 + a_1 \cdot (b_1 + 1)$
3	$(1,0)$	$c_1 + (a_1 + 1) \cdot b_1$
4	$(1,1)$	$c_1 + (a_1 + 1)(b_1 + 1)$

That is, Bob sets its choice bit to be $2a_2 + b_2 + 1$, and obtains the value $c_1 + (a_1 + a_2) \cdot (b_1 + b_2)$. Bob sets its share c_2 to be equal to the value it has received in the OT_1^4.

Claim 5 *Suppose that* OT_1^4 *is secure. Then, above protocol securely computes the random mapping:*

$$(a_1, b_1), (a_2, b_2) \mapsto (c_1, c_2)$$

where c_1, c_2 *are random bits subject to the constraint* $c_1 + c_2 = (a_1 + a_2) \cdot (b_1 + b_2)$.