0368.4170: Cryptography & Game Theory Ran Canetti and Alon Rosen

Lecture 4 - General secure two party and multi party computation

11 November 2009 Fall 2009

Scribes: O. Weisse and D. Sotnikov

1 Overview

In this lecture we will discuss secure two party and multi party computation. Assume we have m parties. Each party P_i holds a private value x_i, and all parties want to compute some agreed upon function $f(x_2, x_2..., x_m)$. We will discuss and sketch the security requirements from such a computation and what difficulties arise from those requirements. We will present a protocol, but will not prove its correctness. (A full proof can be found in *Foundation of Cryptography* - *Volume II Basic Applications*).

2 Toward a Definition

Assume we have m parties. Each party P_i holds a private value x_i, and all parties want to compute some agreed upon function $f(x_2, x_2..., x_m)$. For starters, we would like to have a protocol that allows such computation, that is *correct*. At first correctness seems obvious: the computed output given by the protocol is indeed the right value of $f(x_1, x_2..., x_m)$.

But sometimes it's not that clear what "correctness" means: when we think about it, we cannot require corrupted parties (i.e. parties that deviates from the protocol) to use the input values they received from the outside in the protocol. Thus, we cannot guarantee that the parties which follow the protocol get $f(x_1, x_2..., x_m)$ where $x_1, x_2..., x_m$ are the external inputs. Instead, we will allow the deviating parties to choose new values to be entered into the computation. So correctness is now limited to the right value of $f(x_1, x_2..., x_m)$ for $x_1, x_2..., x_m$ that the parties use as their input values in the protocol.

Another requirement from the desired protocol, is that it is *secret*. That is, at the end of the protocol each party knows the value of $f(x_1, x_2..., x_m)$ and its own value x_i - and *nothing* else.

But now we're running into problems: consider the function $f(x1, x2) = x1 + x2$. That is, each party contributes an (a priori secret) input value, and obtains the exclusive-OR of the two inputs. The protocol instructs P_1 to send its input to P_2; then P_2 announces the result. Intuitively, this protocol is insecure since P_2 can unilaterally determine the output, *after learning P_1's input*. However, this protocol is "correct" in the sense that the output fits the input that P_1 "contributes" to the computation. It also preserve secrecy since for the given function, XOR, the parties don't learn from the protocol anything *more* than they could have learned from the XOR result.

So the "correct function value" depends on the values contributed by the deviating parties. Consequently, the secrecy property depends on these values. This already begins to look bothersome. The solution would be that we will require the process of choosing the inputs by the deviating parties to be done without any knowledge of the inputs of the other parties. So we see here that the two properties "correct" and "secrete" are in some sense intermixed.

In addition we would like to extend the treatment to the case where the expected outputs are taken from some distribution. We will capture this as the task of computing a randomized function, i.e. a function that makes some intended random choices. We will present the random choices of a random function as another input - a random string r. So now our functions looks like this:

$$f(x_1, x_2..., x_m, r)$$

Another issue to note is, how do the parties agree on the random choices the function makes? A party can bias the result value of f by biasing the random choices. A party might also gain some extra information by knowing some prior information about the randomness: for example, a pre-image x for some one way function h such that $h(x) = r$

In the next section we'll see a definition that addresses all these issues.

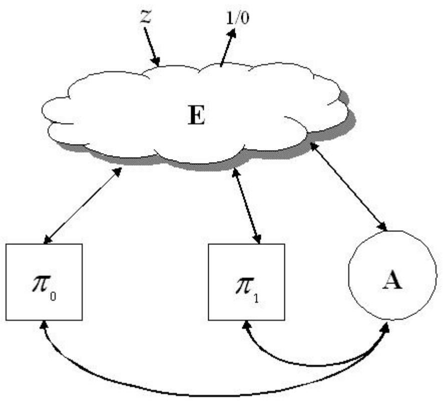
3 Definition of secure computation

We want to define the securely computation as a combination of two properties: secrecy and correctness. The idea is to say that a protocol is secure if running the protocol is "equivalent" to interacting with "ideal system" where both the correctness and the secrecy are "obviously guaranteed". The ideal system which we will consider is one when all parties privately give their inputs to trusted party who locally computes the function and returns the local outputs to the parties. We formalize this idea in three steps:

1. Formalize the real-life setting we're addressing
2. Formalize the ideal scheme
3. Formalize the notion of "behaves like the ideal scheme"

For simplicity we will concentrate on the two party setting.

3.1 Real-life Model:

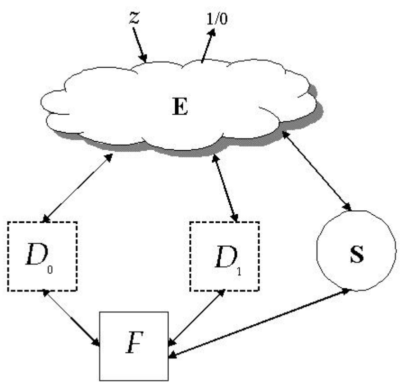


In this model E is an external observer we will call the Environment, π_0 and π_1 are algorithms (ITMs) for two parties in a two-party protocol π, and A is the adversary. We give a high level, short description of the execution of $\pi = (\pi_0, \pi_1)$ together with E and A in the real-life model:

- E gets some input z
- E gives inputs x_0 and x_1 to the π_0 and π_1
- E gives the "Start" massage to the adversary A
- After that A can corrupt one (or zero) of the protocol parties by sending the "corruption" massage

- If the party π_i gets the "corruption" massage it stop to run the protocol and start to execute the adversaries commands. In this way A can change the initial output x_i to another value, to cause to output wrong return value or to deviate from the correct protocol run in another way .
- The parties run the protocol π by sending message to each other. The protocol execution is sequential and at each step only one party is active, it means that two parties can't do anything simultaneously. The adversary A gets all the massages that are sending between the π_0 and $\pi_1$also in the case that they are not corrupted.
- At the end the uncorrupted parties send to E the output values specified by π, the corrupted party can to send any message.
- E outputs one bit.

3.2 The Ideal Model:



In the Ideal Model we don't have the protocol π. Instead we have a trusted party F that gets the input (x_0, x_1) from E, via two "dummy parties" D_0 and D_1, and calculates the $f(x_0, x_1)$, and instead of adversary A we have additional ITM S that simulates the behavior of the adversary. Later we will define the functionality of all the parties at the Ideal Model and

The ideal functionality for computing function f :
$f : (\{0,1\}^\star \times \{0,1\}^\star \times R) \to (\{0,1\}^\star \times \{0,1\}^\star)$

[R denotes the domain of the random inputs]

- gets input x_0 from D_0 and $x_1$from D_1
- gets "corrupt i" message from S
- give x_i to S
- get x'_i from S
- computes $(y_0, y_1) = f(x"_1, x"_2)$ where $x"_i = x'_i$ and $x"_{i-1} = x_{i-1}$
- give y_i to S
- get y'_i from S
- when S says "go j" give $y"_j$ to D_j where $y"_j = \begin{cases} y'_j & i = j \\ y_j & i \neq j \end{cases}$

now we want to define the Security:

First we will give the intuitive definition: the protocol π securely realizes the function f if for any adversary A we can construct some simulator S such that no external observer E can distinguish between the case that he run at the Real-life model or at the Ideal one (that there the computation is done by trusted party and where are no adversary). In this case we say that the real-life scheme behaves like the ideal scheme.

Definition 3.1. (Probability Ensemble) A probability ensemble is an infinite sequence of distributions $\mathfrak{D}= \{D_n\}_{n\in\mathbb{N}}$ where each D_n is a probability distribution. We will usually think of D_n as describing an experiment with security parameter dependent on n.

Definition 3.2. π securely realize F if $\forall PPT\ A\ \exists PPT\ S\ \forall PPT\ E$ the ensembles

$$\{Exec_{\pi,A,En,z}\}_{n\in\mathbb{N},z\in\{0,1\}^{poly(n)}}$$

and

$$\{Exec_{F,S,E,n,z}\}_{n\in\mathbb{N},z\in\{0,1\}^{poly(n)}}$$

are indistinguishable, where $Exec_{\pi,A,E,n,z}$ denotes the output of E at the Real-life Model and the $Exec_{F,S,E,n,z}$ in the Ideal one. Here $n \in \mathbb{N}$ is the security parameter and $z \in \{0,1\}^{poly(n)}$ is the Environment's input.

Remark 3.3. The same frame work (Real-life model vs. The Ideal model) also can be generalized to the case of multiparty computation.

Remark 3.4. If we allow the "corruption" massage to appear only at the beginning of the protocol running we call to such model Static Corruption Model, if "corruption" massage can appear in each round the model is Dynamic Corruption Model.

4 Construction of the Protocol

We will now construct a protocol for joint computation of a potentially randomized function f over GF(2).

We will present the construction as follows:

- Construct the protocol for two parties which honestly follow the protocol, and later examine the protocol transcript, looking for extra information they can learn about the other party's input. We call such parties "honest but curious" parties. (section 4.1)
- Construct a "compiler" that compiles any protocol for "honest but curious" parties to be suitable also for malicious parties. We will run the compiler on our protocol. (section 4.2)
- In section 5 we will expand the protocol for the multi-party case.

4.1 The Protocol for the *Honest but Curious* Parties

4.1.1 Oblivious Transfer Primitive

As a first step toward constructing the protocol, we will now present the Oblivious Transfer primitive and a protocol that implements it:

Assume we have two parties. We denote the first party in this protocol - the *Sender, S*. *S* has n bits of data

$$\{b_1, b_2, ..., b_n\}$$

We denote the second party - the *Receiver, R* . *R* holds $i \in \{1, 2, ...n\}$. *R* wishes to learn b_i.

Requirements:

- Correctness - R will learn the correct value of b_i.
- Secrecy - S will not know which b_i R learned

We will now suggest an implementation. For now, we assume that both S and R are "honest but curious".

A reminder: Asymmetrical encryption: A symmetric encryption scheme consists of three algorithms (G,E,D) with the following properties.

$G = (e, d)$ - the key generation scheme;

E - the encryption algorithm;

D - the decryption algorithm.

We denote e - the public encryption key, and d - the private decryption key. The encryption is given using the public key $cipher = E_e\left(plain\right)$. The decryption is given using the secret private key $plain = D_d\left(cipher\right)$. We assume that semantic security holds even if the public key e is known.

In our case, in addition to all the above, we will require the presence of G' - a variant on the key generation algorithm. G' generates $G' = e'$ in such a way that one cannot distinguish between a given e, generated by the original G, and between e'. Another desired property is that it should be infeasible to decrypt correctly given the random choices of G'.

Example - ElGamal encryption scheme. In this scheme all numbers and arithmetics operations are done in a cyclic group H of prime order q with generator g.

- G - The key generation algorithm: Alice randomly chooses $d = x \in \{1, ..q\}$. She then computes $e = g^x$. Alice sends e to Bob.
- E - the encryption algorithm: Bob wants to encrypt message $m \in H$. Bob randomly chooses $y \in \{1, ..q\}$. He then computes $c = m \cdot e^y = m \cdot g^{xy}$. Bob computes g^y and sends Alice the tuple
$$(c, g^y) = (m \cdot g^{xy}, g^y)$$
- D - the decryption algorithm: Alice computes
$$cipher \cdot (g^y)^{q-x} = m \cdot g^{xy} \cdot g^{yq} \cdot g^{-yx}$$
Since H is of order q we get
$$g^{yq} = (g^q)^y = 1^y = 1$$
Thus we are left with the expression:
$$mg^{xy}g^{-xy} = m$$
- G' - In the case of El-Gamal,, we can set $G' = e' = x \in H$. Since x, g^x have the same distribution in a group of prime order, we get the desired property. Braking the semantic security of (G', E, D) given the random input of G' is as hard as breaking the semantic security of (G, E, D) given only the public key.

The protocol:

We will now use such an asymmetrical encryption scheme (G,E,D) to help us construct the protocol.

1. R uses G to produce (e, d), and a variation of G, G' to produce $n-1$ values $e_1, e_2, ...$ without the decryption keys. R sets $e_i = e$ for his specific i, and sends $\{e_1, e_2, ..., e_n\}$.
2. S computes and sends back
$$\{y_1, y_2 ... y_n\} = \{E_{e_1}\left(b_1\right), E_{e_2}\left(b_2\right), ..., E_{e_n}\left(b_n\right)\}$$
3. R now computes and learn $b_i = D_d\left(y_i\right)$.

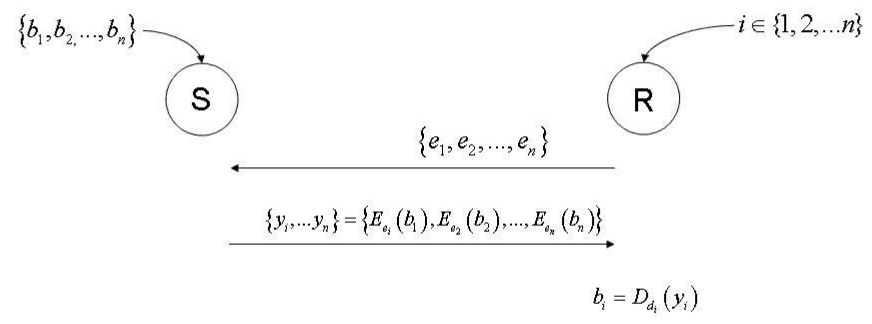

$\{b_1, b_2, \ldots, b_n\}$
$i \in \{1, 2, \ldots n\}$
S
R
$\{e_1, e_2, \ldots, e_n\}$
$\{y_1, \ldots y_n\} = \{E_{e_1}(b_1), E_{e_2}(b_2), \ldots, E_{e_n}(b_n)\}$
$b_i = D_{d_i}(y_i)$

Figure 4.1: Oblivious Transfer protocol

Protocol analysis:

Correctness: R learned bit b_i.

Secrecy: All e_i have the same distributions. Hence, S cannot distinguish between them and cannot know which bit R learned. Assuming R is honest, all $e_j \mid j \neq i$ are random. Thus, he cannot conclude any of the bits $\{b_j | j \neq i\}$ from y_j since he does not have the decryption key d_j.

We're not formally proving security of the protocol. The way to formally prove security is to define an ideal functionality that captures the OT requirements and then show that the protocol realizes this functionality.

4.1.2 Two-Party Secure Computation Protocol

Theorem 1. *Assume Oblivious Transfer protocol exists, for any two party function f there exists a protocol π that securely realizes f for honest but curious parties.*

We will now show a protocol for secure computation of (possibly randomized) functions over GF(2). In the case of GF(2), any function can be presented as a circuit build up from AND (multiplication) & XOR (addition) gates. We denote the two parties participating in this computation P_1, P_2. The inputs for this circuit are $a_1, a_2....a_n$ - for the first party's input, $b_1, b_2, ...b_n$ - for the second party's input, and $r_1, r_2, ...r_k$ - a random string for the random function's coin tosses. See figure 4.2.

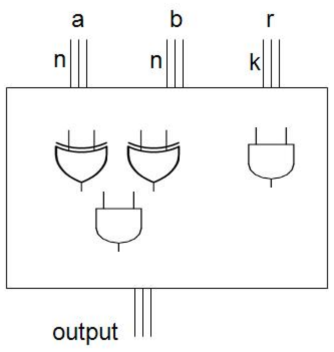


Figure 4.2: GF(2) function represented by a circuit

Computation of any random functionality, which is represented by an arithmetic circuit build up from XOR gates and AND gates, can be done via a private computation with inputs that are shared by two parties. Sharing of bit value v between two parties means a uniformly distributed pair of bits (v_1, v_2)

such that $v = v_1 + v_2$. The first party holds v_1 and the second party holds v_2. Our aim is to propagate, via private computation, shares of the input wires of the circuit to shares of all wires of the circuit, so that finally we obtain shares of the output wires of the circuit. P_1 will hold share w_1 and P_2 will hold his share w_2. The result of the shared computation will be given by $w = w_1 + w_2$.

The protocol:

Step 1 - Sharing the inputs and random values:

Each party splits shares of each of its input bits with the other party. That is, the first party generates

$$\left(a_1^1, a_1^2\right), \left(a_2^1, a_2^2\right) ... \left(a_n^1, a_n^2\right)$$

such that $a_i^1 + a_i^2 = a_i$, and the second party generates

$$\left(b_1^1, b_1^2\right) \left(b_2^1, b_2^2\right) ... \left(b_n^1, b_n^2\right)$$

such that $b_i^1 + b_i^2 = b_i$. That can easily be done by generating a random bit c, and setting $a_i^1 = a_i + c$, $a_i^2 = c$. The first party sends $a_1^2, a_2^2, ..., a_n^2$ to the second party, and the second party sends back $b_1^1, b_2^1, ..., b_n^1$. See figure 4.3.

Since every pair of bits $\left(a_i^1, a_i^2\right), \left(b_i^1, b_i^2\right)$, are uniformly distributed, so that the other party cannot conclude the original bit from his share.

In the case of randomized functions, we will look at the function's coin tosses as another input value - a random string r:

$$f\left(x_1, x_2 ..., x_m, r\right)$$

Each party will locally generate its own share of r - a random string r_i, and will treat this string as a sequence of shares to the corresponding input wires of the circuit.

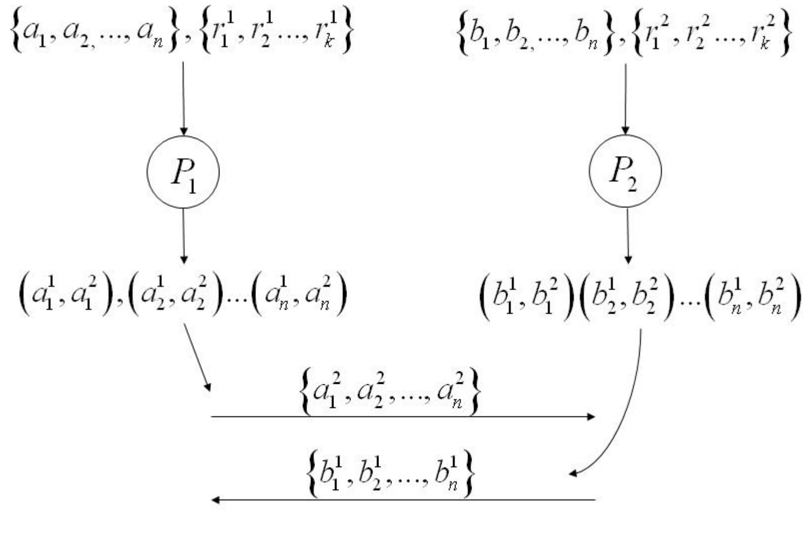

$\{a_1, a_2, ..., a_n\}, \{r_1^1, r_2^1 ..., r_k^1\}$
$\{b_1, b_2, ..., b_n\}, \{r_1^2, r_2^2 ..., r_k^2\}$
P_1
P_2
$(a_1^1, a_1^2), (a_2^1, a_2^2) ... (a_n^1, a_n^2)$
$(b_1^1, b_1^2)(b_2^1, b_2^2) ... (b_n^1, b_n^2)$
$\{a_1^2, a_2^2, ..., a_n^2\}$
$\{b_1^1, b_2^1, ..., b_n^1\}$

Figure 4.3: Input and randomness sharing

Step 2 - Circuit evaluation

Proceeding by some predefined order of the wires, the parties use their shares of the two input wires (u, v) to a gate in order to privately compute shares for the output wire or wires of the gate. For simplicity, we will deal with computation of 2-bit gates. That is: we would like the first party to use u^1, v^1 in order to produce a share of the output w^1, and we would like the second party to use u^2, v^2 to produce its own output share w^2. This is done such that the real output will be given by $w = w^1 + w^2$.

In order to do such private computation we consider two cases:

Evalulation of XOR(addition) gate:

For later use, let us remember that the desired computation result is:

$$w = u + v$$

The first party sets its output to be

$$w^1 = u^1 + u^1$$

The second party sets its output to be

$$w^2 = u^2 + u^2$$

That way the combined true output will be given by:

$$w^1 + w^2 = \left(u^1 + v^1\right) + \left(u^2 + v^2\right)$$

We will now remind ourselves that XOR is commutative in GF(2), thus

$$w^1 + w^2 = \left(u^1 + v^1\right) + \left(u^2 + v^2\right) = \left(u^1 + u^2\right) + \left(v^1 + v^2\right) = u + v = w$$

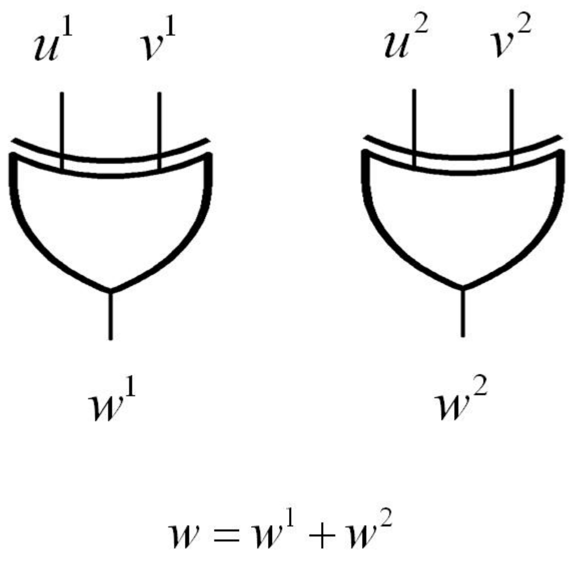


Figure 4.4: Emulation of XOR (adding) gate

Evaluation of a multiplication gate:

The above emulation of the XOR gate was pretty simple: each side used its shares to compute the output as if there was no second party, and and true value was given by simply adding the output shares. Lets try this method for the multiplication gate: P_1 computes $w^1 = u^1 \cdot v^1$. P_2 computes $w^1 = u^2 \cdot v^2$. The result will hopefully be given by:

$$w^1 + w^2 = u^1 \cdot v^1 + u^2 \cdot v^2$$

but since the true result is

$$u \cdot v = \left(u^1 + u^2\right) \cdot \left(v^1 + v^2\right)$$

this method will not give us the desired output.

We will now show how to compute the AND gate result, using 1 out of 4 *Oblivious Transfer*:

For every gate there are two bit inputs u, v, and the result is $u \cdot v$. Each party has a share of each input bit. The first party has u^1, v^1; and the seconds party has u^2, v^2. The true output is given by

$$\left(u^1 + u^2\right) \cdot \left(v^1 + v^2\right)$$

The first party will choose a random bit w^1 and set w^1 as its own share of the output line of the gate. Next it will help the second party to learn the appropriate value w^2. For this , it will prepare

$$w^2_{00}, w^2_{01}, w^2_{11}, w^2_{10}$$

We will now use Oblivious Transfer to give the second party its correct share, without having to know the second party's input shares u^2, v^2. Let us say that

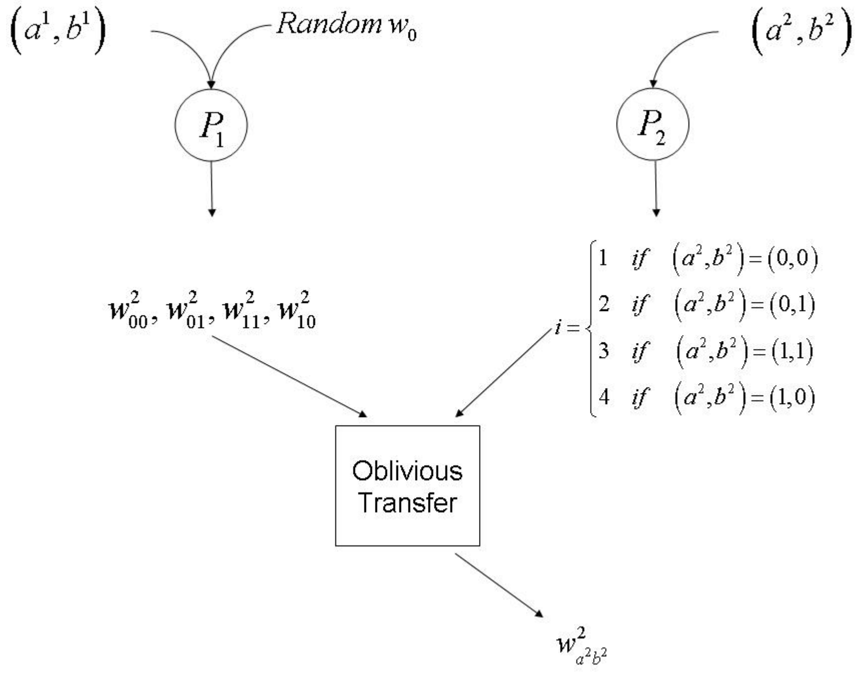
$i = 1$ if $\left(u^2, v^2\right) = (0, 0)$,

$i = 2$ if $\left(u^2, v^2\right) = (0, 1)$,

$i = 3$ if $\left(u^2, v^2\right) = (1, 1)$,

$i = 4$ if $\left(u^2, v^2\right) = (1, 0)$.

Then the second party can use Oblivious Transfer Protocol in order to learn his share of the output, without revealing any information about his input share to the first party. See figure 4.5.



$\left(a^1, b^1\right)$
Random w_0
$\left(a^2, b^2\right)$
P_1
P_2
$w^2_{00}, w^2_{01}, w^2_{11}, w^2_{10}$
$i = \begin{cases} 1 & if \quad \left(a^2, b^2\right) = (0,0) \\ 2 & if \quad \left(a^2, b^2\right) = (0,1) \\ 3 & if \quad \left(a^2, b^2\right) = (1,1) \\ 4 & if \quad \left(a^2, b^2\right) = (1,0) \end{cases}$
Oblivious Transfer
$w^2_{a^2 b^2}$

Figure 4.5: Emulation of multiplication gate, and output sharing using Oblivious Transfer

So what will those values of w^2 will be? This value has to satisfy

$$w^1 + w^2 = u \cdot v = \left(u^1 + u^2\right) \cdot \left(v^1 + v^2\right)$$

For the case of $\left(u^2, v^2\right) = (0,0)$ set

$$w_{00}^2 = w^1 + \left(u^1 + 0\right) \cdot \left(v^1 + 0\right)$$

That way the output value is set by

$$w = w^1 + w_{00}^2 = w^1 + w^1 + \left(u^1 + 0\right) \cdot \left(v^1 + 0\right) = \left(u^1 + 0\right) \cdot \left(v^1 + 0\right)$$

For the case of $\left(u^2, v^2\right) = (0,1)$ set

$$w_{01}^2 = w^1 + \left(u^1 + 0\right) \cdot \left(v^1 + 1\right)$$

That way the true output value is set by

$$w = w^1 + w_{01}^2 = w^1 + w^1 + \left(u^1 + 0\right) \cdot \left(v^1 + 1\right) = \left(u^1 + 0\right) \cdot \left(v^1 + 1\right)$$

For the case of $\left(u^2, v^2\right) = (1,1)$ set

$$w_{11}^2 = w^1 + \left(u^1 + 1\right) \cdot \left(v^1 + 1\right)$$

That way the true output value is set by

$$w = w^1 + w_{11}^2 = w^1 + w^1 + \left(u^1 + 1\right) \cdot \left(v^1 + 1\right) = \left(u^1 + 1\right) \cdot \left(v^1 + 1\right)$$

For the case of $\left(u^2, v^2\right) = (1,0)$ set

$$w_{10}^2 = w^1 + \left(u^1 + 1\right) \cdot \left(v^1 + 0\right)$$

That way the true output value is set by

$$w = w^1 + w_{10}^2 = w^1 + w^1 + \left(u^1 + 1\right) \cdot \left(v^1 + 0\right) = \left(u^1 + 1\right) \cdot \left(v^1 + 0\right)$$

Step 3 - Output Sharing

Each party sends the other party his share of the final output w^i. The output is given by $w^1 + w^2$

4.2 Dealing with malicious faults

In the presence of general faults (also called malicious faults, or Byzantine faults), the protocol for semi-honest parties looks "ridiculously naive", in that it trivially breaks down if the parties deviate even slightly from the protocol. Still, we'll show that it can be transformed or "hardened" into a protocol that withstands malicious faults.

We denote the protocol for "honest but curious parties" π.

Theorem 2. *Assume one way functions exist. Then, there exists a general transformation T such that, for any two party function f and for any polytime protocol π that securely realizes f for semi-honest parties, the protocol $T(\pi)$ securely realizes f for general (Byzantine) faults.*

This theorem suggests that if we have a protocol for *honest but curious* parties — then we can construct a protocol for malicious parties as well using the provided transformation.

We will now define the transformation T mentioned in the theorem.

The idea is to make use of the power of Zero Knowledge proofs. For each message sent in each step, m, there is a witness proving that the message just being sent is in accordance with the protocol. This witness is the secret input, the random tape used for random values, and the messages that have been received so far. Since the computation required to generate each message is polytime the witness is polysize of the message. That means that we have a language $L \in NP$, and every message m which is valid for the protocol satisfies $m \in L$. All we have to do now is make a reduction $L \leq HC$ and run a ZK proof protocol for every m sent.

Before proceeding with the actual transform, let's first do a "warm-up" transformation that deals with the case where the programs run by the parties are deterministic:

The warm-up transformation:

1. Each side commits on its secrete input x_i sending $C_i = COM\left(\rho_i, x_i\right)$. ρ_i is the random string used to commit to x_i.
2. In each step, we will let $\tau = \{C_0, C_1, m_0^{p_1}, m_0^{p_2}, m_1^{p_1}, m_1^{p_2}, ...\}$ be the *Transcript*. I.e. all the messages being sent from P_1 to P_2 and vice versa until now.

   We will define the following language L_i: $\tau \in L_i$ if there exists possible inputs, and random values chosen party P_i such that P_i's messages in τ are consistent with the protocol and the commitments that has been sent. Formally:

$$L_i = \left\{ \tau \mid \begin{array}{l} \exists x_i, \rho_i \text{ s.t. } \rho_i \text{ is the random string used to commit on } x_i; \\ \text{each message sent by } P_i \text{ is the outcom of running } \pi_i \\ \text{on the input } x_i \text{ and the incomming messages } m_0^{p_{3-i}}, m_1^{p_{3-i}} \\ \text{sent by the other party} \end{array} \right\}$$

   In words, there exist input value x_i, and random value ρ_i used for commitment on the input value, such that running the protocol π with those values would produce the messages $m_0^{p_i}, m_1^{p_i}, ...$ being sent so far.

   When in step s party P_i sends a message m to the other party, it has to prove using a Zero Knowledge protocol that the new produced transcript $\tau_s = \tau_{s-1} + m$ satisfies $\tau_s \in L_i$

The full construction:

The above warm-up is not enough: During the protocol each party had to use randomness to produce his messages. We assumed that those random values are uniformly distributed, and that no party has any extra information on those values. This is extremely important e.g. with *oblivious transfer*. The OT protocol relies on the fact that the receiver cannot decrypt anything but the requested bit.

To make sure the random tape is actually random, the first party sends commitments on his random tape $RC_0 = COM\left(\rho_0, r_0\right)$. The other side now sends a sequence of random bits $s_0^0, s_1^0, ...$ The new random tape for the first party, that will be used now on in the protocol, is

$$t_0, t_1, ... \text{ such that } t_i = s_i + r_i$$

That way, even if the first party uses a non-random tape r, the new tape t is as random as s is. This process is now repeated for the second party's random tape.

The transcript now looks like this:

$$\tau = \left\{C_0, C_1, RC_0, RC_1, s_0^0, ..., s_0^1, .., m_0^{p_1}, m_0^{p_2}, m_1^{p_1}, m_1^{p_2}, ...\right\}$$

The language L now looks:

$$L_i = \left\{ \tau \mid \begin{array}{l} \exists x_i, \rho_i^{input} \text{ s.t. } \rho_i^{input} \text{ is the random string used to commit on the input } x_i; \\ \exists r_i, \rho_i^{random-tape} \text{ s.t. } \rho_i^{random-tape} \text{ is the random string used to commit on the} \\ \text{random tape } r_i; \\ \text{Each message sent by } P_i \text{ is the outcom of running } \pi_i \text{ on the input } x_i \\ \text{with the random tape } r_i \text{ and the incomming messages } m_0^{p_{3-i}}, m_1^{p_{3-i}} \\ \text{sent by the other party} \end{array} \right\}$$

In words, there exists input value x_i, with random value ρ_i^{input} used for commitment on this input value, there exist a random tape r_i, with random values $\rho_i^{random-tape}$, used to commit on the random tape - such that running the protocol π with those values would produce the messages $m_0^{p_i}, m_1^{p_i}, ...$ being sent so far.

5 The multi-party case

We will now present a construction for m parties to compute any functionality that is expressed by an arithmetic circuit which consists of XOR gates and multiplication gates. We will assume that there are m parties, and each input consists of n bits.

Our construction follows the same outline presented in the two-party case. We will show a construction for the *honest but curious*, and then present a compiler for generating a protocol for malicious parties.

5.1 Multy Party Computation for the *honest but curious* Parties

Now we assume that each party holds the values of some of the input lines to the circuit, and in addition there are random input lines.

Our aim is to propagate, via private computation, shares of the input wires of the circuit to shares of all wires of the circuit, so that finally we obtain shares of the output wires of the circuit.

Step 1 - Sharing the inputs:

Each party i splits and shares each of its input bits with all other parties. That is, for every $j = 1, ...n$ and $k \neq i$ the party selects a random bit r_j^k and sends it to party k as its share of the i-th input. i-th's own share of the j-th bit will be $\sum \left(r_j^k\right) + x_j$. That way, the true bit value will be given by XOR-ing all the shares. Notice that since the shares are all random bits - the distribution of up to $n-1$ shares is uniformly distributed in $\{0,1\}$ and independent of the real input value. The random inputs to the function are treated as in the two-party case. That is, each party locally sets its share of each random input wire to a freshly chosen random bit.

Step 2 - Circuit Evaluation:

Proceeding by the order of wires, the parties use their share of the two input wires to a gate in order to privately compute shares for the output-wire of the gate. Suppose that the parties hold shares for the two input-wires of some gate: that is, for $i = 1, ...m$, party i holds the shares a_i, b_i, where $a_1...a_m$ are the shares of the first wire, and $b_1...b_m$ are the shares of the second wire. We consider the usual two cases:

Evalulation of XOR(addition) gate:

Each party sets its output wire to be $a_i + b_i$. That way, the shared output will be given by

$$\sum (w_i) = \sum (a_i + b_i)$$

Since addition (XOR) is commutative under GF(2) it gives us:

$$\sum (w_i) = \sum (a_i + b_i) = \sum (a_i) + \sum (b_i) = a + b$$

and that is the desired true output.

Evaluation of a multiplication gate:

This is where life becomes complicated. The true value is given by

$$\left(\sum_{i=1}^{m} a_i\right)\left(\sum_{i=1}^{m} b_i\right) =$$

$$\sum_{i=1}^{m} a_i \cdot b_i + \sum_{\substack{i \neq j \\ 1 \leq i,j \leq m}} a_i \cdot b_j =$$

$$\sum_{i=1}^{m} a_i b_i + \sum_{1 \leq i < j \leq m} (a_i b_j + a_j b_i)$$

Thus, if every party i will have his share set to

$$w_i = a_i b_i + \sum_{i < j \leq m} (a_i b_j + a_j b_i)$$

then the true output will be given by $\sum w_i$.

Since each party i knows a_i, b_i it is left with the task of finding out the value of $a_i b_j + a_j b_i$ for every $i < j$. The solution: For each i, j, it is j-party responsibility to help the i-th party learn $a_i b_j + a_j b_i$ for every i-party that holds $i < j$. Of course, we do not want them to share their inputs. The idea is using *Oblivious Transfer* the same way we did with the two party case: The j party will compute $a_i b_j + a_j b_i$ for every possible combination of (a_i, b_i). There are four such combinations. It will transfer the correct combination result to i-party using Oblivious Transfer Protocol. That way, the i-th party will learn the j-th share it needs in order to compute its own share. Nothing else can be learned due to the Oblivious Transfer security.

Step 3 - Output Reconstruction

For each output wire, we denote A - the group containing all the parties that needs to know the value of this specific output wire, by circuit specification. Every party sends its output share w^i to all other parties in A. The output of the specific wire is given by

$$\sum_{i \in A} w_i$$

5.2 Compiler for the multi-party case

The compiler idea is basically the same as in the two-party case but with few extensions. We want to avoid errors even for the case of a case of a deviating coalition (i.e., a colluding set of deviating parties). We will require that every communication between any two parties will be done on a bulletin board. If one of the parties is not satisfied with what it is getting from some other party, he posts it on the bulletin board so all other honest parties will know that something is wrong. This is done to make sure that all parties have a consistent global view of the computation.

1. Each party commits to its secrete input x_i sending $C_i = COM\left(\rho_i, x_i\right)$. We will use non interactive commitments (ie, the commitment stage is one message sent by the committer) so a single commitment message is used to commit to the input in front of all other parties.

2. Randomness generation - a natural thing to do here is have each party send the other parties $s_0^i, s_1^i, ..$ as we did in the two-party case. But wait! There is a problem here: after $m-1$ parties have posted their randomness shares, the m-th party has the luxury of deciding what the combined randomness would be, by setting his s^i. The correct way would be to have everyone put commitments on their randomness share s^i on the bulletin board - and then have everyone decommit on these values. That way, no party can "change his mind" about his randomness share.

   At the beginning of the protocol, assume P_i wants to set his random string. P_i chooses a random value r_i and sends commitment on this value on the bulletin board. Every party P_j s.t. $j \neq i$ chooses s_i^j and sends commitment on this value on the bulletin board. After all parties have committed they all decommit, except for P_i. Then P_i set his random string to be

   $$t_i = r_i + \sum_{j \neq i} s_i^j$$

   That way, even if one of the parties, or a coalition of parties uses a non-random tape r, the new tape t is as random as long as one of the s^j is random. From now on, every time one of the parties needs to use randomness, the witness for each random value t_i is r_i and ρ_i used to commit on r_i.

3. In each step, for every two-party conversation between party i and j we will let

   $$\tau = \{m_0^{p_0}, m_0^{p_1}, ...m_1^{p_0}, m_1^{p_1}, ...\}$$

   be the *Transcript*. I.e. all the messages being sent by all parties until now.

   The language L now looks:

   $$L_i = \left\{ \tau \mid \begin{array}{l} \exists x_i, \rho_i^{input} \text{ s.t. } \rho_i^{input} \text{ is the random string used to commit on the input } x_i; \\ \exists r_i, \rho_i^{random-tape} \text{ s.t. } \rho_i^{random-tape} \text{ is the random string used to commit on the} \\ \text{random tape } r_i; \\ \text{Each message sent by } P_i \text{ is the outcom of running } \pi_i \text{ on the input } x_i \text{ with the} \\ \text{random tape } r_i \text{ and the incoming messages } m_0^{p_0}, m_0^{p_1}, ...m_1^{p_0}, m_1^{p_1}, ... \\ \text{sent by all the other parties} \end{array} \right\}$$

   In words, there exists input value x_i, with random value ρ_i^{input} used for commitment on this input value, there exist a random tape r_i, with random values $\rho_i^{random-tape}$, used to commit on the random tape - such that running the protocol π with those values would produce the messages $m_0^{p_i}, m_1^{p_i}, ...$ being sent so far, in response to the messages received from the other parties as specified in τ.

   When in step s a party sends a message m to the other party, it has to prove using Zero Knowledge protocol that the new produced transcript $\tau_s = \tau_{s-1} + m$ satisfies $\tau_s \in L_i$

A full proof of this protocol appears in *Foundation of Cryptography - Volume II Basic Applications* [1].

References

[1] Oded Goldreich, *Foundation of Cryptography - Volume II Basic Applications*. Weizmann Institute of Science, Cambridge University Press, 2004.