600.641 Special Topics in Theoretical Cryptography 3/20/2007

Lecture 14: Secure Multiparty Computation

Instructor: Susan Hohenberger *Scribe: Adam McKibben*

1 Overview

Suppose a group of people want to determine who has the highest salary. One method would be for each person to reveal their salary to the others. But what if no one wanted the others to know their actual salary? Another method might be to have each party reveal their salary to some trusted party. The trusted party could then determine who had the highest salary and tell everyone the corresponding name. However, each player must still reveal their salary to a trusted party (whom they *all* have to trust), which may not exist in a real world situation. Is there a way to perform this computation without requiring a trusted party?

This is the kind of problem that secure multiparty computation was invented to solve. The problem was first addressed by Yao [Yao82] and a method for performing secure multiparty computation[1] was developed by Goldriech, Micali and Wigderson in [GMW87]. Today we will be covering the GMW results.

1.1 Goals of secure MPC

Let's begin by informally trying to define what the security guarantees for multiparty computation (MPC) should be. Suppose there are several parties $P_1, P_2, ..., P_n$ who each have some private input $x_1, x_2, ...x_n$ and wish to perform some polynomial-time computation $f(x_1, x_2, ...x_n, R) = (y_1, y_2, ..., y_n)$ where R is some randomness and $(y_1, y_2, ..., y_n)$ are private output values for each party.

The protocol (π) for this computation should be:

- correct - π should allow the parties to correctly compute f
- privacy - for $P_1, P_2, ..., P_n$, each player's input remains private. That is, no one learns any more about their input than can be deduced from the output.
- output delivery - the protocol doesn't end until everyone receives an output (the output may indicate failure)
- fairness - if one party gets the answer, so does everyone else (either everyones gets the answer, or the output for everyone is failure)

[1]with an honest majority

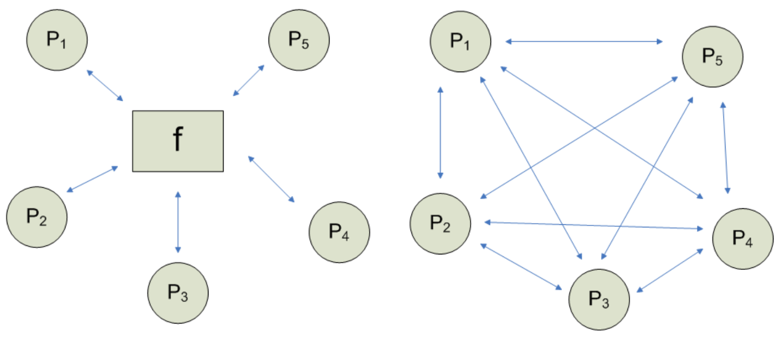


Figure 1: Ideal world vs. Real world

1.2 Formalizing Security

In an ideal world the parties would each send their input to the function and receive their output. This requires a trusted party or machine to calculate f. In the real world, we have no trusted party. The parties must calculate the function by exchanging messages with each other using π.

In defining security, we consider the following game. Suppose there is an environment who can give all the parties inputs, communicate with any corrupted parties, and see all the parties outputs. This enviroment's job is to try to decide whether it is in the real or ideal worlds. We say that a protocol π is secure if the environnment cannot distinguish these two worlds non-negligibly better than guessing.

For a good protocol, no environment, which can give all the parties inputs and communicate with any corrupted parties should be able to distinguish between the real and ideal worlds.

This means that in the ideal world, we must show that there is an "ideal adversary" denoted S who controls the same parties in the ideal world as the real adversary A does in the real world. The goal is to show that anything A can learn (about the private inputs of honest parties) by running π, the ideal adversary S can also learn by interacting with the ideal functionality f. In other words, since S can't do any real damage, then neither can A. Consider that here the only real thing the simulator can do is change the inputs it gives to f.

Definition: Let π be an n party protocol. Then π t-securely computes f if:

$$\forall A \ \exists S \text{ such that } \forall I \subseteq [n] \text{ with } |n| \leq t \quad REAL_{\pi,A,I} \approx IDEAL_{f,S,I}$$

where I is a set of indices of the corrupted parties.

There are some disappointing impossibility results for how large t may be. For example, if more than $\frac{1}{2}$ the parties are corrupt, then we *cannot* achieve all four of our informal security goals simultaneously. In GMW [GMW87], we will be able to acheive all four of the security goals by assuming an honest majority (i.e., $t < \frac{n}{2}$).

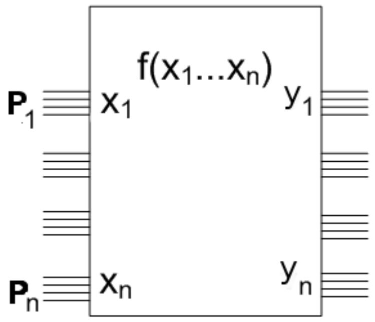


Figure 2: Circuit for f

2 Strategy

Suppose we have some function $f(x_1, x_2, ...x_n) = (y_1, y_2, ..., y_n)$ that can be computed in polynomial time. (For now, we drop the randomness R. The interested reader should think about how we might add randomness to the strategy detailed herein.)

Instead of modeling the computation as a Turing machine, [GMW87] decided to model the function as a circuit. This circuit has some inputs $x_1, x_2, ...x_n$ which are controlled by parties $P_1, P_2, ..., P_n$ respectively, and some outputs $(y_1, y_2, ..., y_n)$. (Figure 2)

Any circuit can be constructed using only AND and NOT gates. So we assume without loss of generality that these are the only gates in the circuit. We also assume that maximum the number of inputs for each gate is two.

The strategy that [GMW87] used was this:

1. Design a protocol with honest but curious (HBC) parties. HBC parties will follow the protocol, but will collect all the information to which they have access and try to compute some additional information from it.

2. Compile the HBC protocol into a protocol that can be used when the majority of the participants are honest. The remaining parties may cheat, collude or not follow the protocol and the honest parties will still receive their correct output and have privacy.

3 Setup

Suppose there are n parties. Each party is assigned an "identity" from 1 to n and we assume that each party knows the identity of all other parties. Each P_i knows their input x_i, which may be a string. We will be working in $\mathbb{F}_2$, so all computations are modulo 2. *Remember* we are in the HBC model, so each party will follow these instructions!

Each P_i shares each bit b of their input with all the other parties as follows:

1. Choose n random bits $b_1, ..., b_n$ such that $b = \sum_{i=1}^{n} b_i \bmod 2$.

2. Send share b_j to P_j (including one share to one's self).

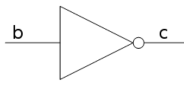
Now the parties start evaluating the circuit together from left to right, top to bottom. Each gate must be evaluated so that each party owns a share of the input and of the output. At each step of the evaluation, all the shares should be uniformly distributed (i.e the value for the share belonging to any party should be equally likely to be 0 or 1.) If at some step the share owned by one party is likely to be 0 or 1, the other parties could continue the computation without that bit.

At the very end of the protocol, all the shares of the output wires representing y_1 should be sent to P_1. The same should be done for $P_2 ... P_n$. But we are getting ahead of ourselves, let's first see how we securely, but privately evaluate these gates.

4 Evaluation

4.1 NOT gates

A NOT gate has an input b and an output $c = b + 1$. The input $b = \sum b_j$ (the sum of shares b_i held respectively by party P_i) and we want one output $c = \sum c_i$, where each share c_i is held by party P_i.



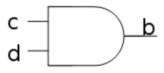
When a NOT gate is reached it can be evaluated as follows:

- P_1 sets $c_i = b_i + 1 \mod 2$
- $\forall i = 2, ..., n, P_i$ sets $c_i = b_i$

Which party is chosen as P_1 is unimportant. It is only necessary that one of the parties be designated as P_1.

4.2 AND gates

An AND gate has two inputs c, d and one output b. When working mod 2, an AND gate is equivalent to $b = cd$.



AND

c	d	b
0	0	0
0	1	0
1	0	0
1	1	1

Each party P_i holds input shares c_i, d_i and wants to obtain an output share b_i, where the following holds:

$$b = \sum_{i=1}^{n} b_i = \left(\sum_{i=1}^{n} c_i\right)\left(\sum_{i=1}^{n} d_i\right) = \sum_{i=1}^{n} c_i d_i + \sum_{1 \le i \le j \le n} (c_i d_j + c_j d_i)$$

Each party knows it own $c_i d_i$, but each $(c_i d_j + c_j d_i)$ part must be computed together by parties P_i and P_j.

Define:

$$b_i = \sum_{j=1}^{n} b_{ij}$$

where

$$b_{ii} = c_i d_i$$

and

$$b_{ij} + b_{ji} = c_i d_j + c_j d_i \text{ for } i \neq j$$

In other words, consider breaking each share (b_i) of b further into shares (b_{ij}).

Since we are working mod 2, we can move b_{ji} to the other side of the equation.

$$b_{ij} = b_{ji} + c_i d_j + c_j d_i$$

Now we need to describe how each party obtains their shares b_{ij} of their output share b_i. This process is not symmetric. That is, each pair of parties P_i and P_j must execute a small protocol together to obtain b_{ij} and b_{ji}, respectively. Here's how they do it. Suppose $j > i$.

1. P_j wants b_{ji}.
2. P_j picks b_{ji} at random from $\{0, 1\}$.
3. P_i wants $b_{ij} = b_{ji} + c_i d_j + c_j d_i$.
4. P_i knows c_i and d_i.
5. P_j knows c_j, d_j and b_{ji}.

P_i would like to learn the value of $b_{ji} + c_i d_j + c_j d_i$ without revealing its values for c_i and d_i. Similarly, P_j does not want to reveal its values for c_j and d_j. P_i and P_j can use $\binom{4}{1}$OT to accomplish this. In this exchange, P_j is the sender and P_i is the receiver. P_j will offer $< b_{ji}, b_{ji} + c_j, b_{ji} + d_j, b_{ji} + c_j + d_j >$ and P_i will select which one of these to receive based on the values of c_i and d_i as shown below.

P_i knows		P_i wants
c_i	d_i	b_{ij}
0	0	b_{ji}
0	1	$b_{ji} + c_j$
1	0	$b_{ji} + d_j$
1	1	$b_{ji} + c_j + d_j$

The actual value for

$$b = \sum b_i = \sum_i^n \sum_j^n b_{ij}$$

.

Thus, we've seen how to evaluate both NOT and AND gates, and we are done!

As the final step, each party must send its shares of y_i to P_i.

5 Compiling HBC protocol for honest majority world

The protocol shown above works only if all the parties are honest but curious. If one or more parties cheat, the whole protocol will break down. For example, one party could compute everything according to the protocol until a certain point, then decide to change its inputs or begin sending random values.

So how do we make this protocol work with only an honest majority world (where $t < \frac{n}{2}$)? Where:

- Cheaters can collude and share inputs freely
- Cheaters know the "code" of honest parties (which follow the protocol)
- Cheaters know who the other cheaters are
- Cheaters do not know honest party inputs or random coins

Suppose there is a Turing machine for P_1 which has an input tape, a random tape, and a work tape and where the other parties can write to its inputs. If we know each party's input and randomness, then everything else is deterministic. If everyone follows the protocol, the outcome can only go one way.

We want to stop any party from changing its inputs part way through the computation. To do this we make each party commit to the values on its tapes: commit(input, randomness, 0^k). Each time a party changes some value of its tapes (eg. writing to its work tape) it must make a new commitment to the change and provide a zero knowledge proof that this change is the valid one given its former state. (For example, if P_1 with share b_i is passing through a NOT gate, then he must commit to his output $c_i = b_i + 1$ and prove that commit(c_i) is correctly computed from commit(b_i).)

These commitments are never actually opened because doing so would reveal private information. It is sufficient to have the zero knowledge proof that the commitment is correct. Now if a party tries to change some value midway, it will not be able to give the zero knowledge proof. These proofs must happen between every two parties, at each step of the computation.

Even with these changes the protocol is still broken with respect to privacy and fairness. The problem is in the initial commit step. If some parties commit to randomness which isn't actually random, then the other cheaters can know each others randomness and the final outputs may be skewed. And how would one *prove* that some bits were randomly chosen anyway??

So we want to fill the random tape of each party P_i with real random bits, but we can't allow others to know what they are. We also can't allow any party to control its own randomness. Let's consider how we do this for party P_1.

To get the first bit for the random tape of P_1, each party commits to a bit. Then all parties, except for P_1 decommit their bits. P_1 calculates the first bit by XORing all the bits, including its own and writes this to the random tape. P_1 won't decommit but must prove that it actually wrote the correct XORed value to the tape. The commitment scheme used must be non-malleable. This must be done for each bit of the random tape of each party.

There are additional problems remaining with this protocol. Each of them must be fixed individually. For example, one party could follow the protocol until the last step. Then it would refuse to send its shares to the proper parties. The cheater would receive its result, but no one else would, breaking fairness.

To fix this problem, a secret sharing scheme could be added to the protocol (such as Shamir's). This would divide the original inputs x_i for each party up among the others. This would allow a number of parties $> t$ to reconstruct the input of a cheater, but any coalition of t or less members would not be able to reconstruct the inputs. (Recall that t is the max number of corrupt parties tolerated.) If some party P_i tries to quit early, the remaining parties could reconstruct its input. They could then restart the protocol without the cheater and simulate its behavior based on the reconstructed input.

6 What if we aren't sure if majority is honest?

This is a great question. But we are out of time! See the recent work of Katz [Kat07] for more.

References

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. *19th ACM Symposium on the Theory of Computing*, pages 218–229, 1987.

[Kat07] Jonathan Katz. On achieving the "best of both worlds" in secure multiparty computation. In *(to appear) STOC '07*, 2007. Available at http://eprint.iacr.org/2006/455.

[Yao82] A.C. Yao. Theory and applications of trapdoor functions. *23rd Annual Symposium on Foundations of Computer Science*, pages 80–91, 1982.