**CS 276 – Cryptography** January 26/28, 2016

# Lecture 3,4: Multiparty Computation

*Instructor: Sanjam Garg* *Scribe: Joseph Hui*

## 1 Constant-Round Multiparty Computation

Last time we considered the GMW protocol, which gives us an algorithm for securely evaluating any circuit $C(x_1, x_2, ..., x_n)$ where each party $p_i, i \in \{1...n\}$ has a part of the input $x_i$.
One issue with this protocol is that it may take a very large number of rounds, dependent on the depth of the circuit, because each gate must be evaluated in a topological order. Let's consider some ideas to fix this problem.

### Currying

Let's consider one simple idea that doesn't work. If we have a circuit $C(x_1, x_2, ..., x_n)$, we can curry the inputs to get a new circuit, say $C'$, that takes in only one input and itself outputs a circuit. Then the next circuit also takes one input and outputs a circuit, and so on, so that $C'(x_1)(x_2)(x_3)...(x_n) = C(x_1...x_n)$. So Alice passes Bob a garbled version of $C'(x_1)$; Bob passes Carol a garbled version of $C'(x_1)(x_2)$; etc. However, this approach cannot possibly work. The reason is that Bob can just simulate Carol, David, etc. in his head, and thus compute $C(x_1, x_2, ..., x_n)$ for any values of $x_3...x_n$ that he might choose.

### Rounds independent of circuit depth

Now let's consider an idea that will reduce the round complexity, from depending on circuit depth to independent of circuit depth. The GMW protocol evaluates a circuit $C(x_1, ...x_n)$ and takes a number of rounds proportional to the circuit depth of $C$. $C$ may be very deep, but can we find some other, shallower circuit that we can feed into the GMW protocol, that nevertheless allows us to perform the desired computation?
Indeed, we can. Instead of computing the value $C(x_1, ...x_n)$ itself, we will securely compute a garbled version of $C$; and each party will be able to use the garbled $C$ to compute the desired value.
Recall Yao's protocol. We defined a ppt algorithm, Garble, such that $\text{Garble}(C, \omega, x)$ for some randomness $\omega$ and input $x$ gives us a garbled circuit $\tilde{C}$ (syntactically, a set of encrypted bitstrings and a translation table). $\tilde{C}$, combined with the appropriate labels for $x$ and some input $y$, can then be used to evaluate $C(x, y)$. In particular, we can let $y$ be empty. Then this gives a garbled circuit $\tilde{C}$ that, when combined with the appropriate labels for $x$, produces $C(x)$.
Consider the algorithm $D(\cdot, \cdot) = \text{Garble}_C(\cdot, \cdot)$ which actually performs this garbling operation. It takes as input some randomness, and some concatenated inputs $x_1, x_2, ..., x_n$. Actually, let's say it takes in $\omega_1...\omega_n$ and computes $\omega = \oplus_i \omega_i$ to use as its randomness (so that if any player is uncorrupted, the randomness will be truly random). It outputs a garbled circuit $\tilde{C}$ and some labels - encryption keys for each input wire. Now, note that $D$ can be parallelized - the encryptions for each gate can be computed completely independently of every other gate, so the circuit depth does not grow with the circuit depth of $C$. After all, what does $D$ do? First it has to XOR all the randomness together - this takes circuit depth $O(n)$ (or even $O(\log n)$). Now its only job is to use

some encryption keys (which it reads off from the randomness) to encrypt some other encryption keys (ditto). Each key is encrypted twice, as before. So the circuit depth is $O(\log n)$, plus the circuit depth of two encryptions, and it is independent of $C$. Note that the encryption circuit may increase in depth with the security parameter.
Let's review. We know how to use GMW to evaluate a circuit $C$, but it takes rounds proportional to the circuit depth $C$. Instead of $C$, we consider another circuit $D$. $D$ has circuit depth independent of $C$ (dependent only on the number of players and the security parameter). Each player generates randomness $\omega_i$, and we use the GMW protocol to securely compute $D(\omega_1, x_1, ..., \omega_n, x_n)$, that is, $\tilde{C} \circ \text{labels}_{x_1,...,x_n}$. Each player now has access to the garbled circuit $\tilde{C}$ as well as the appropriate input labels; and each player simply evaluates the garbled circuit using the input labels, thus computing the answer $C(x_1, ...x_n)$.
Security follows easily. The GMW guarantees that each player learns nothing except the output, i.e. $\tilde{C}$ plus the input labels. And, in turn, Yao's protocol guarantees that using $\tilde{C}$ and the input labels, each player can learn nothing except the output. So each player learns nothing except the output, and we are done.

## Constant rounds: BMR protocol

The above protocol still has round complexity dependent on the security parameter. We would like constant round complexity (assuming the number of players is constant). We will use the same idea as before: use GMW, but instead of computing the answer, compute a garbled circuit, which in turn may be used to compute the answer.
The difference is in the encryption that the garbled circuit will use. Before, we used a generic encryption circuit, whose depth might grow with the security parameter. Now, we use a specific encryption scheme which we will define, and this will also require us to change our protocol in some ways.
Recall the simple "one-time pad" PRG encryption scheme: Alice and Bob have a shared key $k$ and a PRG $G$; and Alice simply sends a message $m \oplus G(k)$, whereas Bob XORs the message with $G(k)$ to recover it. We will use this as our encryption scheme. Now, we must be extremely careful about what computation we secure with the GMW protocol. Including any sort of complex computation, such as the computation of $G(k)$, will increase circuit depth. So, we will take great pains to keep every possible computation out of GMW; the only thing that GMW will do is to securely combine keys for us by XOR-ing them together.
The garbled circuit structure is similar to Yao's construction. We have a circuit with some gates and wires. Each wire has four keys: two 0-keys and two 1-keys, which we call $p_0, q_0, p_1, q_1$ respectively. Each gate has four encryptions, one for each possible output. Suppose we have a gate with input wire $\alpha$ with value 1, and input wire $\beta$ with value 0, and output wire $\gamma$ (which happens to be an AND gate, so it has value $1 \wedge 0 = 0$). Then we XOR one of $\alpha$'s 1-keys, one of $\beta$'s 0-keys, and one of the encryptions. Whether we will use $p$ or $q$ depends on the encryption; we will use $pp, pq, qp, qq$ once each. The reason we have two 0-keys ($p$ and $q$) instead of just one, as in Yao's construction, is because we are using the XOR operation, and reusing keys would allow an attacker to discover correlations between the encrypted messages. This gives us the 0-keys for the output wire $\gamma$ (actually it gives us a "seed key", which we can use to generate the appropriate keys).
Now, let's describe the protocol.

1. First, each player samples two keys and a "blind bit" for each wire (either by just sampling

them randomly, or by sampling a $\kappa$-bit string $s$ and using it as the seed of a PRF). This gives $\left(k_0^\omega(i), k_1^\omega(i), \lambda_i^\omega\right)$.
The "seed key" for a wire $\omega$ with bit $b$ is the concatenation of each player's keys, $\text{key}_b^\omega = k_b^\omega(1) \circ \ldots \circ k_b^\omega(n)$, and the blind bit for a wire $\omega$ is the XOR of each player's blind bits, $\lambda^\omega = \lambda_1^\omega \oplus \ldots \oplus \lambda_n^\omega$.
The blind bit is necessary because each player will eventually learn the correct computation on the garbled circuit, which includes knowing which of her keys was used on each wire. If she knows how her keys correspond with 0 and 1, then she could figure out the value of each wire. The blind bit will be used to randomize the relationship, so that this does not happen.

2. Each player actually generates keys. Specifically, it uses the PRG to generate $p_b^\omega(i), q_b^\omega(i) = G\left(k_b^\omega(i)\right)$, the two keys for bit $b$ on wire $\omega$. Each key is length $n\kappa$ for a total of $2n\kappa$ bits.

3. The players run a GMW protocol to compute four encryptions for each gate. As with the previous construction, each gate can be parallelized. Considering a gate $g$ with input wires $\alpha, \beta$ and output $\gamma$, each player feeds the values $\left(p_b^\alpha(i), q_b^\alpha(i)\right), \left(p_b^\beta(i), q_b^\beta(i)\right)$; $\lambda_i^\alpha, \lambda_i^\beta, \lambda_i^\gamma$; $k_b^\gamma(i)$ into the protocol. That is, she feeds in the 0- and 1-keys for the input wires, all the blind bits, and her keys for the output wire. The protocol then computes $\oplus p_{\lambda^\alpha \oplus x}^\alpha(i) \oplus p_{\lambda^\beta \oplus y}^\beta(i) \oplus \text{key}_{g(x,y)\oplus\lambda^\gamma}^\gamma$ for $x, y = 0, 0$; then the same with $p, q$ for $x, y = 0, 1$, then with $q, p$ for $x, y = 1, 0$, and finally with $q, q$ for $x, y = 1, 1$. That is, for each possible value on the input wires, it computes the XOR of the appropriate keys with the output wire's key corresponding to the correct output value, all with the appropriate blinding.
How is such a computation carried out? Given specific values of $\alpha, \beta, \gamma, x, y$, the circuit can compute this value by simply performing $O(n)$ XOR operations on the blind bits, keys, and the values $k_b^\gamma(i)$. Thus it can be carried out with constant circuit depth and the GMW protocol can be carried out in a constant number of rounds.

4. Each player broadcasts $\lambda_i^\omega, k_b^\omega(i)$ for everyone else's input wires. Now, if we consider (say) a wire $\omega$ that corresponds to player 1's (Alice's) input, she has been sent $\lambda_i^\omega$ for all $i \in 2\ldots n$. In addition, she knows $\lambda_1^\omega$ because she generated it. So she knows the value of $\lambda_1^\omega$. On the other hand, player 2 does not know this value, because she does not know the value of $\lambda_1^\omega$; Alice did not send it to her.

5. Now each player knows the blind bits for her input wires alone. Since she also knows the actual value of her input, she knows which key should be used in the computation. Thus, she broadcasts the keys for her own input wires. Specifically, if the $j$-th bit of player $i$'s input goes on wire $\omega$, she broadcasts $k_{(x_i[j])\oplus\lambda^\omega}^\omega$.

This concludes the protocol.

## Lecture 4: IT-Secure Multiparty Computation

We have now seen protocols for secure multiparty computation. However, these protocols relied on computational hardness assumptions. In fact, we can construct protocols that are information-theoretically secure, even against unbounded adversaries. The drawback of this approach is that we will not be able to guarantee security if almost all parties are corrupted by the adversary. We will be able to guarantee security only if at most some constant fraction is corrupted.

One protocol, the CCD protocol, accomplishes this task, but we will not be looking at it as it only guarantees statistical security. Instead, we look at the BGW protocol, which guarantees perfect security against $t$ adversaries. In the semi-honest case, it can protect against $t$ adversaries as long as $t < \frac{n}{2}$; in the malicious case, as long as $t < \frac{n}{3}$. A result by Rabin and Ben-Or also provides a construction against $t < \frac{n}{2}$ malicious corruptions, assuming the presence of a broadcast channel, and providing statistical security.
The basic idea that we will use is secret sharing. A secret sharing scheme works as follows. A dealer has a bit $b$, and she wants to give "shares" $a_1 ... a_n$ to players $p_1 ... p_n$ such that the players can recover $b$ as long as they all come together and share their values of $a_i$. This is an $n$-out-of-$n$ scheme, since all $n$ players need to cooperate; we can also define $t$-out-of-$n$ schemes, in which only $t$ players need to cooperate.
Note that we used this same idea in GMW. GMW had three stages:

1. Secret sharing: each party acts as a dealer and secret-shares a bit $b$ for each of its input wires, by dealing out bits $a_i$ such that $\sum_i a_i = b$.

2. Computation: For each gate, the parties securely combined their shares for the input wires to produce shares for the output wire.

3. Reconstruction: All parties revealed their shares for the output wire to recover the desired result.

In the BGW protocol, we will use a $t$-out-of-$n$ secret-sharing scheme, called Shamir's Secret Sharing. The scheme works as follows. Consider a field $F$ (e.g. the integers modulo a prime) with $|F| > n$, and $n$ distinct nonzero values $\alpha_1, \alpha_2, ..., \alpha_n \in F$. Sample a random polynomial $p(x)$ of degree $t$ such that $p(0) = s$, the secret to be shared. Now given $t + 1$ points of the polynomial, we can uniquely determine the polynomial and evaluate it at 0 to recover the secret (this is covered in e.g. CS70). Given $t$ points of the polynomial, every value of the secret is equally likely and thus we gain zero information about it.
Now that we've decided on how to share secrets, let's consider the computation phase. We have a function to evaluate, which can be expressed as a circuit composed of addition and multiplication gates (over the field). We'll consider a gate, with two inputs, $p(0)$ and $q(0)$, encoded in two randomy polynomials $p$ and $q$ with shares $\alpha_i, \beta_i$ split among the $n$ players. Suppose this gate is an addition gate. Then, we want to compute the shares of a third polynomial, $r$, such that $r(0) = p(0) + q(0)$, and $r$ is a random polynomial among all degree-$t$ polynomials with this property. This is simple: our output polynomial is $p + q$ with $(p + q)(0) = p(0) + q(0)$, and each player computes her new output share as $\gamma_i = \alpha_i + \beta_i$. $p + q$ is clearly a random polynomial (being the sum of two random polynomials) and has the desired evaluation at 0. Similarly, if this gate is a constant-multiplication gate (i.e. we want to multiply the polynomial by a fixed constant $c$), evaluation is also easy; everyone multiplies their share by $c$.
How about multiplication of $p(0)$ and $q(0)$? At first, this may seem easy: $pq$ is a new polynomial that indeed satisfies $(pq)(0) = p(0) q(0)$, so perhaps each player can just multiply their two shares. However, this doesn't work, for two reasons. First of all, the product of two random polynomials is not necessarily itself random (the distribution may be nonuniform). Secondly, the resulting polynomial will be of degree $2t$, not degree $t$.
We can fix both of these problems. First, each player $i$ multiplies her two shares, $f_a(\alpha_i)$ and $f_b(\alpha_i)$, together. Now the players have shares corresponding to the polynomial $pq$ (note that here is where

we use the assumption that $2t < n$; if not, then the players do not have enough points to uniquely determine $pq$, and the information is lost).

Now, let's fix the randomization problem through a process called "rerandomization". This is easy: we can just add a random polynomial $r$ of degree $2t$, such that $r(0) = 0$. Any polynomial plus a random polynomial is a random polynomial, so this fixes our problem, and does not change the encoded secret. In order to add this polynomial, one party will be the dealer: she will pick the polynomial at random, generate the shares, and give them to all parties, who will then add the assigned shares to their shares. And since no one party can be trusted to be the dealer, we have every party deal in turn, in total adding $n$ polynomials to our original polynomial. This produces a new polynomial which we will call $h(x)$.

How will we fix the degree problem? We will devise a procedure to truncate our polynomial $h(x)$. Clearly, if we have a random polynomial of degree $2t$, chopping off the highest $t$ coefficients gives us a polynomial of degree $t$ that is also random. All parties collectively hold shares $h(\alpha_1)...h(\alpha_n)$ of $h(x)$, and they want to compute $\hat{h}(\alpha_1)...\hat{h}(\alpha_n)$ of $\hat{h}(x) = \text{truncate}_t(h(x))$. This can be described as a matrix operation: $\begin{pmatrix} \hat{h}(\alpha_1) \\ ... \\ \hat{h}(\alpha_n) \end{pmatrix} = V_\alpha \cdot P_T \cdot V_\alpha^{-1} \begin{pmatrix} h(\alpha_1) \\ ... \\ h(\alpha_n) \end{pmatrix}$. $V_\alpha$ is the matrix that evaluates a degree-$n$ polynomial, with its coefficients listed in a vector, at points $\alpha_1...\alpha_n$. That is, $V_\alpha = \begin{pmatrix} 1 & \alpha_1 & ... & \alpha_1^{n-1} \\ ... & & & ... \\ 1 & \alpha_n & ... & \alpha_n^{n-1} \end{pmatrix}$ (verify in your head that this corresponds to polynomial evaluation).

Correspondingly, $V_\alpha^{-1}$ is the inverse of $V_\alpha$, so it converts a polynomial from point-value representation to coefficient representation. Finally, $P_T$ is the truncation matrix: it is an $n$-by-$n$ matrix with a $t+1$-by-$t+1$ identity matrix in the top left corner, and the rest all zeroes. So, we have $h$ in point-value form; we multiply it by $V_\alpha^{-1}$ to get the coefficient form; we multiply it by $P_T$ to chop off all but the lowest $t+1$ coefficients, reducing the degree; and then we multiply it by $V_\alpha$ to put it back in coefficient representation. This produces the shares of $\hat{h}(x)$ that we want.

However, how do we actually perform this computation, without sharing information and thus leaking the answer? Observe that we expressed the computation as a matrix multiplication. This means that the computation is necessarily linear; that is, we can express it as $\hat{h}(\alpha_i) = c_{i1}h(\alpha_1) + c_{i2}h(\alpha_2) + ... + c_{in}h(\alpha_n)$ for some known constants $c_{ij}$. Then how do we compute this? Well, this computation consists solely of addition and constant-multiplication operations, which we already know how to do! So, whenever we need to do a multiplication operation, we need to do degree reduction, and we can bootstrap it by running a subprotocol that requires only additions/constant multiplications. Note that at the end of this subprotocol, players do not reveal their shares publicly; they only send their shares to the player that requires them (i.e. player $i$ requires the shares for $\hat{h}(\alpha_i)$). Now that we know how to perform addition and multiplication, we can carry out the evaluation of the circuit, and this completes the protocol.

## Malicious IT-Secure Multiparty Computation

Now let's try to extend this scheme to the malicious setting. When players can act maliciously, they have many ways to break the protocol. In the computation stage, they can do all sorts of crazy things: send random garbage, send a nonzero polynomial in the rerandomization phase, etc. In the input stage, they could generate a polynomial that is of degree $> t$, so that different sets of

players see a different polynomial. In the reconstruction stage, they could send incorrect shares.
First, let's think about how to solve the issue of sending incorrect shares in the reveal stage. Let's assume that up to this point, everyone has acted honestly. So we now have a polynomial of degree $t$, and we have $n > 3t$ evaluations of it at various points, one per player. Everyone reveals their evaluation, but the resulting points may not be consistent with any degree-$t$ polynomial. How do we fix this? The answer is simple: we use a Reed-Solomon code, which is an $(n, t+1, n-t)$ code ($n$ symbols, $t+1$ data symbols, $n-t$ minimum distance between codewords) that corrects $\frac{n-t-1}{2}$ errors. When $t < \frac{n}{3}$, this can correct $t$ errors. For example, if $t = 1$ and $n = 4$, so that all of the points lie along a straight line, then I can look at four points and find the line that passes through them, ignoring one point that was corrupted and doesn't lie on the line. In general, the Berlekamp-Welch algorithm can be used to reconstruct the original polynomial.
Secondly, let's think about how to solve the issue of the input. We'll use something called *verifiable* secret sharing. It works as follows:

1. The dealer $D$, wishing to distribute a secret $s$, samples a random polynomial $q(x)$ of degree $t$ such that $q(0) = s$.

2. The dealer samples a random bivariate polynomial $s(x, y)$ of degree $t$ (i.e. with coefficients $c_{tt}x^t y^t + ... + c_{50}x^5 + ...$) such that $s(0, z) = q(z)$.

3. The dealer sends $f_i(x) = s(x, \alpha_i)$ and $g_i(y) = s(\alpha_i, y)$ to player $i$.

4. Each pair of players $i, j$ checks that $f_i(\alpha_j) = g_j(\alpha_i)$. If this is not the case, they complain to the dealer.

Why does this work? Let's think about an intuitive argument first. Let's say the dealer is honest, but a few players are dishonest. When players $i, j$ are communicating, $j$ might give some incorrect value of $g_j(\alpha_i)$. If this happens, $i$ will complain to the dealer, reporting the value that she received. The dealer will then be able to recognize that player $j$ is malicious, and can reveal player $j$'s shares publicly. On the other hand, if the dealer is dishonest and gives players incorrect polynomials, this will trigger a much larger number of complaints among players; the players can then recognize that these complaints could not have arisen from malicious players alone, and conclude that the dealer is dishonest.
Now, let's start working towards a proof. One basic thing we should check is that if all the players are honest, and all of the checks in step 4 pass, then there really is a unique polynomial $s(x, y)$ consistent with all players' shares. That is, a dishonest dealer cannot trick a group of honest players without being detected. Let $K$ be the full set of all honest parties. The basic idea is that if we consider a set $L$ of any $t+1$ parties, their shares fix a specific $s(x, y)$. Now, if their values of $f$ are consistent with the other players' values of $g$, this will imply that $s(\alpha_k, y) = g_k(y)$ for all $k \in K$ (by definition). Similarly, we must have $s(x, \alpha_i) = f_i(x)$ for all $i \in K \setminus L$. This ensures that all parties' shares are consistent with $s$. We will go through the full proof in the next lecture.