6.897: Selected Topics in Cryptography March 13, 2004

Lecture 12

Lecturer: Ran Canetti *Scribe: Dah-Yoh Lim*

1 Recap

Last lecture we started to look at how we could realize any two-party functionality for any number of faults in the $\mathcal{F}_{\mathsf{CRS}}$-hybrid model. In this lecture we will finish this discussion and extend it to the multi-party case. We will also note that we can get rid of the CRS in the case of an honest majority. All the material in this lecture is taken from [CLOS02].

In more detail, we follow the [GMW87] paradigm of first constructing a protocol secure against semi-honest adversaries (i.e., even the corrupted parties follow the protocol specification), then constructing a general compiler that transforms protocols secure against semi-honest adversaries to "equivalent" protocols secure against Byzantine adversaries.

In the previous lecture we presented the ideal oblivious transfer functionality, $\mathcal{F}_{\mathsf{OT}}$. In this lecture we will show how to realize $\mathcal{F}_{\mathsf{OT}}$ for semi-honest adversaries in the plain model, and show how to realize any functionality in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model.

1.1 The Semi-honest Adversarial Model

Recall that there are two natural variants of the semi-honest adversarial model. In the first variant the adversaries can change the inputs of the corrupted parties, but are otherwise passive. In the second variant the environment talks directly with the parties, and the adversaries only listen (cannot even change the inputs). These variants are incomparable, because there are protocols secure under one model but insecure under the other. We will actually need the first variant for the compiler, but the protocol we are going to see shortly is secure under both variants.

1.2 Standard Functionalities

Recall that in lecture 8 we defined "standard functionalities" as the functionalities which do not utilize their direct knowledge of the identities of the corrupt parties. Specifically, it consists of an "outer shell" and a "core". The core is an arbitrary probabilistic polynomial-time algorithm, while the shell is a simple interfacing procedure described as follows. The shell forwards any incoming messages to the core, with the exception that notifications of corruptions of parties are not forwarded to the core. Outgoing messages generated by the core are copied by the shell to the subroutine output tape of the recipient. On top of these, $\mathcal{S}$ is allowed to delay the receiving of inputs and sending of outputs: the shell actually notifies $\mathcal{S}$ about its intentions, and only carries on to do so when $\mathcal{S}$ OKs it. In handling corruptions the shell hands an output "corrupted" to the party that $\mathcal{S}$ wants to corrupt, and hands $\mathcal{S}$ all the I/O history so far of the corrupted party. From now on all of the corrupt party's I/O privileges is taken over by $\mathcal{S}$.

Notice that under this restriction we avoid the problem of not being able to realize functionalities that use the knowledge of the identities of the corrupt parties in an essential way. For example, consider the ideal functionality that lets all parties know which parties are corrupted. Then this

functionality cannot be realized in the face of an adversary that corrupts a single random party but instructs that party to continue following the prescribed protocol.

All the functionalities following are standard ones.

2 Evaluating General Functionalities in the Semi-honest, Two-party case

Here, we follow the [GMW87] paradigm. However, there are two differences. Firstly, [GMW87] considers static adversaries whereas we consider adaptive adversaries. This only makes a difference in the oblivious transfer protocol, but still the proof of security is significantly different. The second difference is that [GMW87] considers function evaluation, whereas we consider more general reactive functionalities where parties may receive new inputs during the protocol execution and each new input may depend on the current adversarial view of the system. This only makes a small difference to the construction.

Since we are talking about standard functionalities in which the shell is merely an interface, we will deal with the core only. Let $\mathcal{F}$ be an ideal two-party functionality and P_0, P_1 be the participating parties.

Preliminary step: Represent the ideal functionality F as a Boolean circuit.

Before we can start to construct a protocol that UC-realizes $\mathcal{F}$, we first represent (the core of) $\mathcal{F}$ via a family $\mathcal{C}_{\mathcal{F}}$ of boolean circuits (the k-th circuit in the family describes an activation of $\mathcal{F}$ when the security parameter is set to k). Following [GMW87], we use arithmetic circuits over GF(2) where the operations are addition and multiplication modulo 2.

There are five types of input lines: inputs of P_0, inputs of P_1, inputs of S, random inputs, and local-state inputs. There are four types of output lines: outputs to P_0, outputs to P_1, outputs to S, and the local state for the next activation.

Step 1: Input sharing. When P_i is activated with new input, it notifies P_{1-i} and does the following: Firstly it shares each input bit b with P_{1-i} by sending $b_{1-i} \leftarrow_R \{0,1\}$ to P_{1-i} and keeping $b_i = b + b_{1-i}$. Secondly, for each random input line, it chooses $r_i \leftarrow_R \{0,1\}$. The shares of the input lines of P_{1-i} are set to 0.

In addition, P_i has its share s_i of each local state line s from the previous activation. (Initially, these shares are set to 0.) P_i's shares of the adversary input lines are set to 0. When P_i is activated by notification from P_{1-i} it proceeds as above, except that it sets its inputs to be 0. Note that at this point, the values of all input lines to the circuit are shared between the parties.

Step 2: Evaluating the circuit. Let a, b denote the input values of the gate, and let c denote the output value. So P_i holds bits a_i and b_i (whereas P_{1-i} holds bits a_{1-i} and b_{1-i}, such that $a = a_i + a_{1-i}$ and $b = b_i + b_{1-i}$). The parties evaluate the circuit gate by gate, so that the output value of each gate is shared between the parties, as follows.

- Addition gates: we want to perform the computation $a + b = c$ in a shared manner, i.e. so that P_i holds c_i and P_{1-i} holds c_{1-i} such that $c_i + c_{1-i} = c$. Since P_i has a_i and b_i, it simply computes $c_i = a_i + b_i$. (Since $a_0 + a_1 = a$ and $b_0 + b_1 = b$, we have $c_0 + c_1 = c$.)

- Multiplication gates: we want to perform the computation $a * b = c$ in a shared manner, i.e. so that P_i holds c_i and P_{1-i} holds c_{1-i} such that $c_i + c_{1-i} = c$. P_0 and P_1 use $\mathcal{F}_{\mathsf{OT}}^4$ as follows: P_0 chooses c_0 at random, and plays the sender with input:

  $$v_{00} = a_0b_0 + c_0, v_{01} = a_0(1 - b_0) + c_0, v_{10} = (1 - a_0)b_0 + c_0, v_{11} = (1 - a_0)(1 - b_0) + c_0.$$

  P_1 plays the receiver with input (a_1, b_1), and sets the output to be c_1. It is easy to verify that indeed $c_0 + c_1 = (a_0 + a_1)(b_0 + b_1)$.

Step 3: Output generation Once all the gates have been evaluated, each output value is shared between the parties. Then:

- P_{1-i} sends to P_i its share of the output lines assigned to P_i.
- P_i reconstructs its outputs and outputs them.
- P_i keeps its share of each local-state line (and will use it in the next activation).
- Outputs to the adversary are ignored [1].

Theorem 1. *Let $\mathcal{F}$ be a standard ideal functionality. Then the above protocol realizes $\mathcal{F}$ in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model for semi-honest, adaptive adversaries.*

Note that the theorem holds unconditionally. In fact, it holds even if the environment and the adversary are computationally unbounded. (But of course, computational assumptions are required for UC-realizing the oblivious transfer functionality.)

Proof. (very rough sketch): For any real-life adversary $\mathcal{A}$, we construct an ideal adversary $\mathcal{S}$ that fools all environments $\mathcal{Z}$. In fact, the simulation will be unconditional and perfect (i.e., $\mathcal{Z}$'s views of the two interactions will be identical):

- The honest parties obtain the correct function values as in the ideal process.
- P_0 sees only random shares of input values, plus its outputs. This is easy to simulate.
- P_1 receives in addition also random shares of all intermediate values (from $\mathcal{F}_{\mathsf{OT}}$). This is also easy to simulate.
- Upon corruption, it is also easy to generate local state.

□

Remarks: There is a protocol [Yao86] that works in a constant number of rounds. The theorem can be proven for static adversaries, and it also works for adaptive adversaries with erasures.

Open: what about adaptive adversaries without erasures? Is there a general construction with constant number of rounds in this case?

[1] thus, we effectively prevent the ideal-model adversary from utilizing its capability of sending and receiving messages. This simplifies the construction, and only strengthens the result

3 Protocol Compilation

Now that we have a protocol secure against semi-honest adversaries we move on to constructing a general compiler that transforms protocols secure against semi-honest adversaries to "equivalent" protocols secure against Byzantine adversaries.

3.1 Review of the GMW Compiler

We want to force the malicious parties to follow the protocol specification. Let us first briefly review the GMW compiler. In summary, the GMW compiler has three components: input commitment, coin-tossing and protocol emulation (where the parties prove that their steps are according to the protocol specification):

- Parties commit to inputs.
- Parties commit to uniform random tapes using secure coin-tossing, which ensures uniformity. Note that a simple coin-tossing protocol in which both parties receive the same uniformly distributed string is not sufficient here. This is because the parties' random tapes must remain secret. Instead, an augmented coin-tossing protocol is used, where one party receives a uniformly distributed string (to be used as its random tape) and the other party receives a commitment to that string.
- Protocol emulation: Run the original protocol Q, and in addition the parties use zero-knowledge protocols to prove that they follow the protocol. That is, each message of Q is followed be a ZK proof of the NP statement [2]:

  > "There exist input x and random input r that are the legitimate openings of the commitments I sent above, and such that the message I just sent is a result of running the protocol on x, r, and the messages received so far".

  The key point is that, due to the soundness of the proofs, even a malicious adversary cannot deviate from the protocol specification without being detected. Therefore, in effect the adversary is limited to semi-honest behavior by the commitments to the random tape and to the inputs. Furthermore, since the proofs are zero-knowledge, nothing "more" is revealed in the compiled protocol than in the basic protocol. We thus conclude the security of the compiled protocol (against malicious adversaries) directly from the security of the basic protocol (against semi-honest adversaries).

3.2 Constructing a UC "GMW Compiler"

The naïve approach to solution is to take the same compiler, but use universally composable commitments, coin-tossing, and zero-knowledge as sub-protocols.

The problem with this naïve idea is that if ideal commitment is used, there is no commitment string to prove statements on! This is because the receiver of a universally composable commitment

[2] Observe that a protocol specification is a deterministic function of a party's view consisting of its input, random tape and messages received so far. Further observe that each party holds a commitment to the input and random tape of the other party and that the messages sent so far are public. Therefore, the assertion that a new message is computed according to the protocol is an NP statement (and the party sending the message knows an adequate NP-witness to it)

receives no information about the value that was committed to. (Instead, the recipient receives only a formal "receipt" assuring it that a value was committed to.) Thus, there is no NP-statement that a party can prove relative to its input commitment. This is in contrast to the GMW protocol where standard (perfectly binding) commitments are used and thus each party holds a string that uniquely determines the other partys input and random tape.

To get around this problem, first observe that in GMW the use of the commitment scheme is not standard. Specifically, although both parties commit to their inputs etc., they never decommit. Rather, they prove NP-statements relative to their committed values. Thus, a natural primitive to use would be a "commit-and-prove" functionality, which is comprised of two phases. In the first phase, a party "commits" (or is bound) to a specific value. In the second phase, this party proves NP-statements in zero-knowledge relative to the committed value. This notion is implicit in the work of [GMW87], and was also discussed by Kilian [Kil89]. We formulate this notion in a universally composable commit-and-prove functionality, denoted $\mathcal{F}_{\mathsf{CP}}$, and then use this functionality to implement all three phases of the compiler.

3.2.1 The Commit and Prove Functionality $\mathcal{F}_{\mathsf{CP}}$

The Commit and Prove functionality, $\mathcal{F}_{\mathsf{CP}}$ (which is parameterized by a relation R), with committer C, receiver V, and an adversary S:

- Upon receiving $(sid, C, V, \text{"}commit\text{"}, w)$ from (sid, C), add w to the list W of committed values, and output $(sid, C, V, \text{"}receipt\text{"})$ to (sid, V) and $\mathcal{S}$.
- Upon receiving $(sid, C, V, \text{"}prove\text{"}, x)$ from (sid, C), send $(sid, C, V, x, R(x, W))$ to $\mathcal{S}$. If $R(x, W)$ then also output (sid, x) to (sid, V).

Note that V is assured that the value x it received in step 2 stands in the relation with the list W that C provided earlier, and C is assured that V learns nothing in addition to x and $R(x, W)$. As in the case of $\mathcal{F}_{\mathsf{ZK}}$, the $\mathcal{F}_{\mathsf{CP}}$ functionality is defined so that only correct statements (i.e., values x such that $R(x, w) = 1$) are received by V in the prove phase. Incorrect statements are ignored by the functionality, and the receiver V receives no notification that an attempt at cheating in a proof took place (this is done for simplicity; error messages could have been added instead).

We now show how to UC-realize $\mathcal{F}_{\mathsf{CP}}$ in the $\mathcal{F}_{\mathsf{ZK}}$-hybrid model, i.e. in a hybrid model with ideal access to an ideal zero-knowledge functionality, $\mathcal{F}_{\mathsf{ZK}}$ [3]. Essentially, in the commit phase of the commit-and-prove protocol, the committer commits to its input value w using some standard commitment scheme, and in addition it proves to the receiver, using $\mathcal{F}_{\mathsf{ZK}}$ with an appropriate relation, that it "knows" the committed value. In the prove phase, where the committer wishes to assert that the committed value w stands in relation R with some public value x, the committer presents x and w to $\mathcal{F}_{\mathsf{ZK}}$ again- but this time the relation used by $\mathcal{F}_{\mathsf{ZK}}$ asserts two properties: first that $R(x, w)$ holds, and second that w is the same value that was previously committed to.

Specifically, if we have COM, a perfectly binding, non-interactive commitment scheme, then the protocol for realizing $\mathcal{F}_{\mathsf{CP}}^{R}$ in the $\mathcal{F}_{\mathsf{ZK}}$-hybrid model is as follows:

- To commit to w, (sid, C) computes $a = COM(w, r)$, adds w to the list W, adds a to the list A, adds r to the list R, and sends $(sid, C, V, \text{"}prove\text{"}, a, (w, r))$ to $\mathcal{F}_{\mathsf{ZK}}^{R_c}$, where

$$R_c = \{(a, (w, r)) : a = COM(w, r)\}.$$

[3] Functionality $\mathcal{F}_{\mathsf{ZK}}$ expects to receive a statement x and a witness w from the prover. It then forwards x to the verifier, together with an assertion whether $R(x, w)$ holds, where R is a predetermined relation.

- Upon receiving $(sid, C, V, a, 1)$ from $\mathcal{F}_{\mathsf{ZK}}^{R_c}$, (sid, V) adds a to the list A, and outputs $(sid, C, V, \text{“}receipt\text{”})$.
- To give x and prove $R(x, W)$, (sid, C) sends $(sid, C, V, \text{“}prove\text{”}, (x, A), (W, R))$ to $\mathcal{F}_{\mathsf{ZK}}^{R_p}$, where

$$R_p = \Big\{ \big((x, A), (W, R)\big) : W = w_1, ..., w_n, A = a_1, ..., a_n, R = r_1, ..., r_n, R(x, W), \\ \text{and } a_i = COM(r_i, w_i) \text{ for all } i \Big\}$$

- Upon receiving $(sid, C, V, (x, A), 1)$ from $\mathcal{F}_{\mathsf{ZK}}^{R_p}$, (sid, V) verifies that A agrees with its local list A, and if so outputs (sid, C, V, x).

Theorem 2. *The above protocol realizes $\mathcal{F}_{\mathsf{CP}}$ in the $\mathcal{F}_{\mathsf{ZK}}$-hybrid model for non-adaptive adversaries (assuming the security of COM).*

Proof. For any $\mathcal{A}$, construct an $\mathcal{S}$ that fools all $\mathcal{Z}$. As usual, $\mathcal{S}$ runs $\mathcal{A}$. There are two different cases to analyze depending on which party is corrupted.

In the case of a corrupted committer, intuitively $\mathcal{S}$ must be able to extract the decommitment value w from $\mathcal{A}$ during the commit phase of the protocol simulation. This is because, in the ideal process, $\mathcal{S}$ must explicitly send the value w to $\mathcal{F}_{\mathsf{CP}}$ (and must therefore know the value being committed to). Fortunately, this extraction is easy for $\mathcal{S}$ to do because $\mathcal{A}$ works in the $\mathcal{F}_{\mathsf{ZK}}$-hybrid model, and any message sent by A to $\mathcal{F}_{\mathsf{ZK}}$ is seen by S during the simulation. In particular, S obtains the ZK-proof message sent by A to $\mathcal{F}_{\mathsf{ZK}}^{R_c}$, and this message is valid only if it explicitly contains the decommitment.

In the commit phase $\mathcal{S}$ obtains from $\mathcal{A}$ the message $(sid, C, V, \text{“}prove\text{”}, a, (w, r))$ to $\mathcal{F}_{\mathsf{ZK}}^{R_c}$. If R_c holds (i.e. $a = COM(w, r)$) then $\mathcal{S}$ inputs $(sid, C, V, \text{“}commit\text{”}, w)$ to $\mathcal{F}_{\mathsf{CP}}^{R}$. In the prove phase, $\mathcal{S}$ obtains from $\mathcal{A}$ the message $(sid, C, V, \text{“}prove\text{”}, (x, A), (W, R))$ to $\mathcal{F}_{\mathsf{ZK}}^{R_p}$. If R_p holds then $\mathcal{S}$ inputs $(sid, C, V, \text{“}prove\text{”}, x)$ to $\mathcal{F}_{\mathsf{CP}}^{R}$.

In the case of a corrupted verifier, in the commit phase, $\mathcal{S}$ obtains from $\mathcal{F}_{\mathsf{CP}}^{R}$ a $(sid, C, V, \text{“}receipt\text{”})$ message, and simulates for $\mathcal{A}$ the message (sid, C, V, a) from $\mathcal{F}_{\mathsf{ZK}}^{R_c}$, where $a = COM(0, r)$ (this commitment to 0 instead of the corresponding witness is the only difference between simulated and real executions). In the prove phase, $\mathcal{S}$ obtains from $\mathcal{F}_{\mathsf{CP}}^{R}$ a $(sid, C, V, \text{“}prove\text{”}, x)$ message, and simulates for $\mathcal{A}$ the message $(sid, C, V, (x, A))$ from $\mathcal{F}_{\mathsf{ZK}}^{R_p}$, where A is the list of simulated commitments generated so far.

Analysis of $\mathcal{S}$: in the case of a corrupted committer, the simulation is perfect (this is due to the fact that the commitment is perfectly binding [4]). In the case of a corrupted verifier, the only difference between the simulated and real executions is that in the simulation the commitment is to 0 rather than to the witness. Thus, if $\mathcal{Z}$ distinguishes then we can construct an adversary that breaks the hiding property of the commitment. □

The above protocol fails in the case of adaptive adversaries (even erasing will not help; if the randomness used to commit to some w is erased after the commitment is generated, then later the committer cannot prove that the commitment really corresponds to w). But one can prove adaptive security given that the commitment COM is equivocal, i.e. a simulator (having access to

[4] if the commitment were not binding, then w in the commit phase versus the prove phase could be different, defeating the whole purpose of defining this commit and prove primitive, which was to prove (in ZK) statements regarding the committed values.

some trapdoor information) can generate a commitment c such that given any w at a later stage, it can find some randomness r_w such that $c = COM(w, r_w)$, and therefore open the commitment in both ways.

Whether $\mathcal{F}_{\mathsf{CP}}$ can be realized *unconditionally* in the $\mathcal{F}_{\mathsf{ZK}}$-hybrid model is still an open problem.

3.2.2 The Compiler in the $\mathcal{F}_{\mathsf{CP}}$-hybrid Model

Informally, the protocol compiler uses the "commit" phase of $\mathcal{F}_{\mathsf{CP}}$ in order to execute the input and coin-tossing phases of the compiler. The "prove" phase of $\mathcal{F}_{\mathsf{CP}}$ is then used to force the adversary to send messages according to the protocol specification and consistent with the committed input and the random tape resulting from the coin-tossing. The result is a universally composable analog to the GMW compiler. We remark that in the $\mathcal{F}_{\mathsf{CP}}$-hybrid model the compiler is unconditionally secure against adaptive adversaries, even if the adversary and the environment are computationally unbounded.

Let $P = (P_0, P_1)$ be a protocol. (Intuitively, we think of P as a protocol designed to work against semi-honest adversaries. But formally P is any arbitrary protocol.) Construct the compiled protocol $Q = C(P)$. Protocol Q uses two copies of $\mathcal{F}_{\mathsf{CP}}$, where in the i-th copy Q_i is the prover. Q_0 Proceeds as follows: (Q_1's code is analogous.)

1. Committing to Q_0's randomness (done once at the beginning): Q_0 chooses random r_0 and sends $(sid.0, Q_0, Q_1, \text{“}commit\text{”}, r_0)$ to $\mathcal{F}_{\mathsf{CP}}$. Q_0 receives r_1 from Q_1, and sets $r = r_0 + r_1$.

2. Committing to Q_1's randomness (done once at the beginning): Q_0 receives $(sid.1, Q_1, Q_0, \text{“}receipt\text{”})$ from $\mathcal{F}_{\mathsf{CP}}$ and sends a random value s_0 to Q_1.

3. (a) Receiving the i-th new input, x: Q_0 sends $(sid.0, Q_0, Q_1, \text{“}commit\text{”}, x)$ to $\mathcal{F}_{\mathsf{CP}}$.

   (b) In addition, let M be the list of messages received so far. Q_0 runs the protocol P on input x, random input r, and messages M, and obtains either a local output value, in which case it simply output this value, or an outgoing message m, in which case it sends $(sid.0, Q_0, Q_1, \text{“}prove\text{”}, m)$ to $\mathcal{F}_{\mathsf{CP}}$, where the relation is

$$R_P = \Big\{ \big((m, M, r_1), (x, r_0) \big) : m = P_0(x, r_0 + r_1, M) \Big\}.$$

4. Receiving the i-th message, m: Q_0 receives $(sid.1, Q_1, Q_0, \text{“}prove\text{”}, (m, M, s_0))$ from $\mathcal{F}_{\mathsf{CP}}$. It verifies that s_0 is the value sent in Step 2, and that M is the set of messages sent to Q_1. If so, then run P_0 on incoming message m and continue as in Step 3(b).

Theorem 3. *Let P be a two-party protocol. Then the protocol $Q = C(P)$, run with Byzantine adversaries, emulates protocol P, when run with semi-honest adversaries. That is, for any Byzantine adversary $\mathcal{A}$ there exists a semi-honest adversary $\mathcal{S}$ such that for any $\mathcal{Z}$ we have:* $\mathsf{Exec}_{\mathcal{P},\mathcal{S},\mathcal{Z}} \approx \mathsf{Exec}^{\mathcal{F}_{\mathsf{CP}}}_{\mathcal{Q},\mathcal{A},\mathcal{Z}}$

Corollary 4. *If protocol P securely realizes $\mathcal{F}$ for semi-honest adversaries then $Q = C(P)$ securely realizes $\mathcal{F}$ in the $\mathcal{F}_{\mathsf{CP}}$-hybrid model for Byzantine adversaries.*

Proof. The proof is omitted here, but it is pretty straightforward. Both the theorem and the corollary hold for the case of adaptive adversaries and for the case of static adversaries. The proof is unconditional and the simulation is perfect. It is essential in the proof that $\mathcal{S}$ be able to change the inputs to parties. (Indeed, recall that we use the variant of the semi-honest model where the adversary can change the inputs.) □

4 Extension to the Multiparty Case

We now describe how the two-party construction above is extended to the setting of multi-party computation, where any number of parties may be corrupt. Recall that in this setting, each set of interacting parties is assumed to have access to an authenticated broadcast channel. The outline of our construction is as follows. Similarly to the two-party case, we first construct a multi-party protocol that is secure against semi-honest adversaries (as above, this protocol is essentially that of GMW). Then, we construct a protocol compiler (again, like that of GMW), that transforms semi-honest protocols into ones that are secure even against malicious adversaries. This protocol compiler is constructed using a one-to-many extension of the commit-and-prove functionality, denoted $\mathcal{F}_{\mathsf{CP}}^{1:M}$. (In the one-to-many extension, a single party commits and proves to many receivers/verifiers.) The extension of the protocol that UC-realizes two-party $\mathcal{F}_{\mathsf{CP}}$ to a protocol that UC-realizes one-to-many $\mathcal{F}_{\mathsf{CP}}^{1:M}$ involves one-to-many extensions of $\mathcal{F}_{\mathsf{ZK}}$ and $\mathcal{F}_{\mathsf{COM}}$ as well.

4.1 Extension to the Multiparty Case: The Semi-honest Protocol (fixed set of n parties)

Here we consider the case where the set of parties participating is fixed, and every party knows each other party. For instance, we could have a preliminary round of "neighbor discovery" in which each party agrees on the participants of the protocol.

The only differences between such a protocol and the protocol for two parties are as follows:

- Each party shares its input among all parties: $x = x_1+, ..., +x_n$.
- Each random input, local state value is shared among all parties in the same way.
- Evaluation of addition gates is done locally by each party as before. Since $(a_1+, ..., +a_n) + (b_1+, ..., +b_n) = (a_1 + b_1)+, ..., +(a_n + b_n)$, the output is shared correctly.
- Evaluation of multiplication gates is done as follows: each pair $i < j$ of parties engage in evaluating the same OT, where they obtain shares c_i, c_j such that $c_i + c_j = (a_i + a_j)(b_i + b_j)$. Each party sums its shares of all the OT's. Iff n is odd then party P_i also adds $a_i b_i$ to the result [5]
- Output stage: All parties send to P_i their shares of the output lines assigned to P_i.

4.2 Extension to the Multiparty Case: Byzantine Adversaries

We have to extend all the functionalities (commitments, ZK, commit and prove) to the case of multiple verifiers (i.e., 1-to-many commitments, ZK, commit and prove). Note that we cannot simply use the two-party analogues, otherwise say a party might be proving different things to different parties.

[5] this is because for a fixed i, as we range over all the pairs $i < j$, we would get $(n-1)-1$ extra terms of the form $a_i b_i + a_j b_j$; iff n is odd then $n-2$ is odd, so P_i has to add an extra $a_i b_i$ to nullify the effect. (if n is even then we need to do nothing. This is so since we are working in GF(2), i.e. adding the same term any even number of times has no effect); $a_j b_j$ will be taken care of by P_j. To see this explicitly, let c_{ij} be the share of the OT obtained by P_i when interacting with P_j and d_i be the end result obtained by P_i (so $d_i = \sum_j c_{ij}$ if n is odd; $d_i = \sum_j c_{ij} + a_i b_i$ if n is even). We want $\sum_i d_i = (\sum_i a_i)(\sum_i b_i)$. But the righthand-side is $\sum_i a_i b_i + \sum_{1 \le i < j \le n} (a_i b_j + a_j b_i) = (1 - (n-1)) \sum_i a_i b_i + \sum_{1 \le i < j \le n} (a_i + a_j)(b_i + b_j) = n \sum_i a_i b_i + \sum_{1 \le i < j \le n} (a_i + a_j)(b_i + b_j)$, which when n is odd simplifies to $\sum_i a_i b_i + \sum_{1 \le i < j \le n} (a_i + a_j)(b_i + b_j)$ and when n is even simplifies to $\sum_{1 \le i < j \le n} (a_i + a_j)(b_i + b_j)$.

Then we have to realize these functionalities, using a broadcast channel (modelled as an ideal functionality, $\mathcal{F}_{\mathsf{BC}}$), which in turn can be realized in an asynchronous network with any number of faults, via a simple "two-round echo" protocol, as follows:

1. When a party P_i gets some input ("*broadcast*", x), it broadcasts the input x to everyone.
2. Upon receiving a value x^j from P_i, party P_j sends the value x^j that it received to all other parties.
3. Party P_j waits to receive a message from every party other than P_i. Denote the message received from P_k by x_k^j. Then, P_j outputs $(broadcast, P_i, x^j)$ iff for every k it holds that $x_k^j = x^j$. Otherwise, it outputs nothing.

Note that the ideal broadcast functionality does not guarantee delivery of messages, nor does it provide any synchrony guarantees for the messages that are delivered. It only guarantees that no two uncorrupted parties in P will receive two different message with the same *sid* (this protocol is analyzed in [GL02].

4.3 Example: The 1:M commitment functionality, $\mathcal{F}_{\mathsf{CP}}^{1:M}$

The first step in realizing $\mathcal{F}_{\mathsf{CP}}^{1:M}$ is to construct one-to-many extensions of universal commitments and zero-knowledge. In a one-to-many commitment scheme, all parties receive the commitment (and the committer is bound to the same value for all parties). $\mathcal{F}_{\mathsf{COM}}^{1:M}$ is defined as follows:

1. Upon receiving $(sid, C, V_1, ..., V_n,$ "*commit*", $x)$ from (sid, C), do:
   - Record x
   - Output $(sid, C, V_1, ..., V_n,$ "*receipt*") to $(sid, V1), ..., (sid, V_n)$
   - Send $(sid, C, V_1, ..., V_n,$ "*receipt*") to $\mathcal{S}$
2. Upon receiving $(sid,$ "*open*") from (sid, C), do:
   - Output (sid, x) to $(sid, V_1), ..., (sid, V_n)$
   - Send (sid, x) to $\mathcal{S}$
   - Halt.

Likewise, in one-to-many zero-knowledge, all parties verify the proof (and they either all accept or all reject the proof). Now, any non-interactive commitment scheme can be transformed into a one-to-many equivalent by simply having the committer broadcast its message to all parties. However, obtaining one-to-many zero-knowledge is more involved, since we do not know how to construct non-interactive adaptively-secure universally composable zero-knowledge. Nevertheless, a one-to-many zero-knowledge protocol can be constructed based on the universally-composable zero-knowledge protocol of [CF01] and the methodology of [G98] for obtaining a multi-party extension of zero-knowledge.

To realize $\mathcal{F}_{\mathsf{CP}}^{1:M}$ in the $\mathcal{F}_{\mathsf{ZK}}^{1:M}$ -hybrid model, we generalize the $\mathcal{F}_{\mathsf{CP}}$ protocol. As with zero-knowledge, this is not straightforward because in the protocol for adaptive adversaries, the $\mathcal{F}_{\mathsf{CP}}$ commit-phase is interactive. Nevertheless, this problem is solved by having the committer commit

to its input value w by separately running the protocol for the commit-phase of (two-party) $\mathcal{F}_{\mathsf{CP}}$ with every party over the broadcast channel. Following this, the committer uses one-to-many zero-knowledge to prove that it committed to the same value in all of these commitments. (Since each party views the communication from all the commitments, every party can verify this zero-knowledge proof.) The prove phase is similar to the two-party case, except that the one-to-many extension of zero-knowledge is used (instead of two-party zero-knowledge). Finally, we note that, as in the two-party case, a multi-party protocol compiler can be constructed in the $\mathcal{F}_{\mathsf{CP}}^{1:M}$-hybrid model in a straightforward way, with no further assumptions.

Lastly, we remark that in the case of an honest majority, we can actually do without the CRS. This is because if we have an honest majority then we can realize $\mathcal{F}_{\mathsf{COM}}^{1:M}$ in the plain model, using known VSS (Verifiable Secret Sharing) protocols, e.g., the ones in [BGW88].

References

[BGW88] M. BenOr, S. Goldwasser, and A. Wigderson. Completeness theorems for noncryptographic fault-tolerant distributed computation. ACM STOC, pages 1–10, 1988.

[CF01] R. Canetti and M. Fischlin. Universally Composable Commitments. In CRYPTO01, Springer-Verlag (LNCS 2139), pages 19V40, 2001.

[CLOS02] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally Composable Two-Party and Multi-Party Secure Computation. In Proc. 34th STOC, pp. 494-503.

[G98] O. Goldreich. Secure Multi-Party Computation. Manuscript. Preliminary version, 1998.

[GL02] S. Goldwasser and Y. Lindell. Secure Computation Without Agreement. In 16th DISC, Springer-Verlag (LNCS 2508), pp. 17-32, 2002. Full version available at http://www.research.ibm.com/people/l/lindell/.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson, How to play any mental game, in Proceedings of the 19th Symposium on the Theory of Computing, New York, 1987, pp. 218-229.

[Kil89] J. Kilian. Uses of Randomness in Algorithms and Protocols. The ACM Distinguished Dissertation 1989, MIT press.

[Yao86] A.C. Yao. How to generate and exchange secrets, in Proceedings of the 27th Symposiumon Foundations of Computer Science, Toronto, Ontario, Canada, 1986, pp. 162V167.