CMSC 858T—Secure Distributed Computation January 26, 2021

Lecture 1

Lecturer: Jonathan Katz *Scribe(s): Deena Postol, Shilpa Roy*

1 Background

Definition 1 A function $f : \mathbb{N} \to \{0,1\}^*$ is **negligible** if for all constants c there exists N such that for $\kappa \geq N$ it holds that $f(\kappa) \leq \frac{1}{\kappa^c}$. ♢

Above, κ is the *security parameter*. Security parameters can represent many things, but one might think of the security parameter as representing the key length.

A *distribution ensemble* $\{X(\kappa, a)\}_{\kappa \in \mathbb{N}, a \in \{0,1\}^*}$ is an infinite set of distributions index by the security parameter and also (optionally) parameterized by some values. Two distribution ensembles are *perfectly indistinguishable* if for all κ, a we have $X(\kappa, a) \equiv Y(\kappa, a)$. They are *statistically indistinguishable* if there is a negligible function ε such that for all κ, a we have $SD(X(\kappa, a), Y(\kappa, a)) \leq \varepsilon(\kappa)$, where the *statistical distance* SD between two distributions X, Y is

$$SD(X, Y) = \frac{1}{2} \sum_r |\Pr[X = r] - \Pr[Y = r]|.$$

Distribution ensembles are *computationally indistinguishable* if for all probabilistic, polynomial-time (PPT) distinguishers D there is a negligible function ε such that for all κ, a, z we have

$$|\Pr[x \leftarrow X(\kappa, a) : D(1^\kappa, a, z, x) = 1] - \Pr[y \leftarrow Y(\kappa, a) : D(1^\kappa, a, z, y) = 1]| \leq \varepsilon(\kappa).$$

The value z here is called *auxiliary input*, and makes the above definition a non-uniform one (since z may depend on κ). In layman's terms, no distinguisher D can differentiate between a sample from X and a sample from Y, even if it knows the string a that these distributions depend on, and even if we give D some extra auxiliary information z.

2 Secure Computation

We are interested in multi-party protocols for computing some (possibly randomized) function f. A randomized function (also called a *functionality*) taking n inputs and producing n outputs means that fixing inputs $x_1, \ldots, x_n$ induces a distribution on the output $(y_1, \ldots, y_n) \leftarrow f(x_1, \ldots, x_n)$. A protocol computing f allows a collection of n parties, holding inputs $x_1, \ldots, x_n$, respectively, to compute f such that each party learns y_i such that, when everyone follows the protocol, the joint distribution on the outputs $y_1, \ldots, y_n$ matches that defined by f.

What would it mean for such a protocol to be secure? At a minimum, we want *correctness* and *privacy*. Correctness intuitively means that the protocol computes f even if some of the parties deviate from the protocol, trying to cause it to fail. This is not trivial to define, since any such definition would have to take into account the fact that deviating parties can always change their inputs (and then run the protocol honestly). Privacy intuitively means that no subset of the parties

"learns anything" about the inputs of parties outside that subset. This, too, is not trivial to define since the evaluation of f itself might leak some information. (Consider the two-party evaluation of the AND function.) Moreover, we want to ensure both correctness and privacy together.

We address all the above by defining security using the *real/ideal paradigm*. That is, we define an "ideal-world" evaluation of f, and then say that a protocol computing f is secure if it is "close" to such an ideal-world execution. In the ideal world, we imagine the parties have access to a trusted third party who evaluates f on their behalf; that is, the parties send their inputs to the trusted third party, who evaluates the function on those inputs and then returns the appropriate output to each party. To formally define security, then, we need to formally define the real-world execution of some protocol, the ideal-world evaluation of f, and what it means for those two to be "close."

2.1 Real-World Execution

Fix some n-party protocol Π. In the real-world execution, the parties are given inputs $x_1, \ldots, x_n$, respectively, and then honest parties run Π as instructed. But there are several details of this execution that need to be specified:

Communication model. How do parties communicate in the course of the protocol? The default model is that any party can send a message to any other party (i.e., the communication graph is a complete graph), but one could consider more restricted communication topologies. Typically, we assume that the communication between any pair of (honest) parties is *authenticated*, meaning that the adversary cannot interfere with or spoof such communication. One might also want to assume that the communication between honest parties is *private*. Note that privacy and message integrity can be ensured using standard cryptographic techniques if the parties can share keys in advance of the protocol execution.

We can also ask what is guaranteed about message delivery. In the *synchronous* communication model, execution of the protocol proceeds in rounds and messages sent in a round are guaranteed to be received by the end of that round. (Here, we typically assume *rushing*, meaning that in each round the corrupted parties receive messages sent to them before that have to decide on what messages they send.) In contrast, in the *asynchronous* model the only guarantee is that a message that is sent is eventually delivered, but messages may arrive out of order or be arbitrarily delayed. (We even allow the adversary to schedule the delivery of messages.) In this class, we will assume the synchronous model unless stated otherwise.

Finally, we may ask whether the parties have access to a *broadcast channel*. This allows any party to send a message to all other parties, such that all other parties are assured that they received the *same* message. (Note that having a party P send the same message to all other parties via independent, point-to-point channels does not achieve this, since if P is cheating then it might send different messages to different parties.)

Adversarial model. Typically, we will have some assumed bound t on the number of parties who might be corrupted. Corrupted parties are assumed to be controlled by a single adversary who can pool all those parties' information together and also control their behavior during the course of the protocol execution. We can consider two classes of adversaries: *semi-honest* parties follow the protocol, but may try to learn disallowed information from the transcript of the protocol execution. *Malicious* parties may deviate arbitrarily from the protocol execution. Other models of adversarial behavior are possible, e.g., a **fail-stop** adversary who follows the protocol except that it may stop the protocol early.

Another question is whether or not we allow the adversary to corrupt parties *adatively*, i.e., during the execution of the protocol, or whether we assume a *non-adaptive* (or *static*) adversary who chooses which parties to corrupt before execution of the protocol begins. In this class we will generally assume a static adversary.

Output distribution. Once we have fixed our model of the real world, execution of a protocol Π in the presence of an adversary $\mathcal{A}$ when the parties hold some specified inputs $x_1, \ldots, x_n$ is well-defined. Let $\text{REAL}_{\Pi,\mathcal{A}}(\kappa, \vec{x}, z)$ be the distribution ensemble representing the vector of outputs of all parties after running protocol Π using security parameter κ and inputs $\vec{x} = (x_1, \ldots, x_n)$ (with party P_i holding x_i) in the presence of an adversary $\mathcal{A}$ corrupting some of those parties and holding input z. Honest parties output whatever the protocol dictates, while corrupted parties output their view of the protocol execution. (A party's view contains its initial input, its randomness, and the messages it received during the protocol execution.)

2.2 Ideal-World Execution

The ideal-world evaluation of a functionality f—where parties have access to a trusted entity computing f on their behalf—proceeds as follows. Each party P_i is given an input x_i. An honest party sends its input to the trusted entity. Some parties may be corrupted (and, as in the real world, corrupted parties are assumed to be controlled by a single adversary $\mathcal{S}$). In the semi-honest model, corrupted parties also send their inputs to the trusted entity; in the malicious model, corrupted parties may change their input and send whatever they like to the trusted entity.[1] The trusted party evaluates f on the inputs it receives (using correctly distributed randomness if f is randomized), and sends the corresponding outputs to each party.[2] Honest parties output whatever they receive from the trusted entity, while corrupted parties can output an arbitrary function of their view (which here includes only their original input and the output they receive from the trusted entity). Let $\text{IDEAL}_{f,\mathcal{S}}(\kappa, \vec{x}, z)$ be the distribution ensemble representing the vector of outputs of all parties in this ideal-world execution using security parameter κ and inputs $\vec{x} = (x_1, \ldots, x_n)$ (with party P_i holding x_i) in the presence of an adversary $\mathcal{S}$ corrupting some of those parties and holding input z.

2.3 Defining Security

With the real world and ideal world defined, we can now define what it means for a real-world protocol Π to securely compute f. Our definition will be to say that Π is secure if any behavior that an (efficient) adversary $\mathcal{A}$ could cause in a real-world execution of Π could also be caused by an (efficient) adversary $\mathcal{S}$ in the ideal-world evaluation of f; since the ideal world is "as secure as we could hope for," this implies security of the real-world execution of Π. More formally (the definition assumes a static corruption model):

Definition 2 Let Π be an n-party protocol computing an n-input functionality f. Π is *t-secure* if for any PPT adversary $\mathcal{A}$ corrupting at most t parties there is an expected[3] polynomial-time adversary $\mathcal{S}$ corrupting the same parties such that the distribution ensembles $\{\text{REAL}_{\Pi,\mathcal{A}}(\kappa, \vec{x}, z)\}_{\kappa\in\mathbb{N},\vec{x},z\in\{0,1\}^*}$ and $\{\text{IDEAL}_{f,\mathcal{S}}(\kappa, \vec{x}, z)\}_{\kappa\in\mathbb{N},\vec{x},z\in\{0,1\}^*}$ are computationally indistinguishable. $\diamondsuit$

[1] In the asynchronous setting, corrupted parties may even refuse to send anything to the trusted entity. We assume a synchronous setting in our current discussion.

[2] In some settings, a malicious adversary may have the ability to *abort* the trusted party and thus prevent honest parties from learning the output; the model must make clear whether this is possible.

[3] This (slight) generalization is necessary for proving efficient protocols secure against malicious adversaries.