CS 600.442 – Modern Cryptography 31 August 2015

Lecture 1: Introduction

Instructor: Abhishek Jain *Scribe: Gabe Kaptchuk and Carlo Olcese*

In this lecture, we will establish some basic definitions that will be used throughout this course. Towards the end, we will make an attempt to formalize the weakest cryptographic primitive, namely, one-way functions.

1 Algorithms and Running Times

Definition 1 (Algorithm) *An algorithm A is a deterministic Turing Machine whose inputs and outputs are strings over an alphabet $\Sigma = \{0, 1\}$.*

Definition 2 (Running Time) *An algorithm A has running time $T(n)$ if for all inputs $x \in \{0,1\}^n$ (i.e., strings of length n over the input alphabet), $A(x)$ halts in time $T(n)$.*

Definition 3 (Polynomial Time) *An algorithm A is a polynomial-time algorithm if $\exists c \in \mathbb{R}$ s.t. A has running time $T(n) = n^c$. Further, if A is polynomial time, then we say that it is* efficient.

Note that here, c could be an arbitrary constant. In particular, it may not be small. Then, does this definition of an *efficient* algorithm reflect what we commonly think of as being efficient? For example, consider $c = 100$. In practice, n^100 may actually be considered "inefficient." For our purposes, however, we will stick with this definition of efficiency. In particular, for us, inefficient algorithms correspond to those that have super-polynomial running times such as $T(n) = 2^n$ or $T(n) = n^{\log n}$.

So far, we have only considered deterministic algorithms. In computer science, and specifically, in cryptography, randomness plays a central role. Therefore, throughout the course, we will be interested in randomized (a.k.a. probabilistic) algorithms.

Definition 4 (Randomized Algorithm) *A randomized algorithm A is a Turing Machine with an additional randomness tape where each bit is chosen uniformly and independently. The output of a randomized algorithm is a distribution.*

Similar to polynomial-time deterministic algorithms, we can also define probabilistic polynomial-time algorithms.

Definition 5 (Probabilistic Polynomial Time) *A randomized algorithm A that has a polynomial running time is referred to as a probabilistic polynomial-time (abbreviated as PPT) algorithm.*

Throughout the class, we will be interested in algorithms that compute functions of our choice. Towards this, we define the notion of function computation.

2 Function Computation

Definition 6 (Function Computation) *A randomized algorithm A computes a function $f : \{0,1\}^* \to \{0,1\}^*$ if $\forall x \in \{0,1\}^*$, A on input x outputs $f(x)$ with probability 1.*

When we say "with probability 1," what exactly is this probability over? It is over the random tape input to the randomized algorithm A.

Remark 1 (Restricting to binary outputs) *Without loss of generality, we can relax the above definition to only consider functions with binary outputs. This is because given f, we can define a sequence of functions $g_1, g_2, \ldots$, where g_i computes the ith output bit of f. Now, we can use A to compute each g_i separately.*

Remark 2 (Relaxing the success probability) *We can further relax the above definition to require A to succeed with probability $1 - \frac{1}{2^{|x|}}$ as opposed to 1. In other words, A is allowed to fail with probability $\frac{1}{2^{|x|}}$. When $|x|$ is large, the failure probability is extremely small, and therefore, A correctly computes f most of the time. This will be sufficient for most applications that we will consider in this course.*

As it turns out, for the case of functions with boolean outputs, we can relax the definition even further to require that A correctly computes f with only probability $\frac{1}{2} + 1p(|x|)$ for some polynomial $p(\cdot)$. Note that computing f with probability $\frac{1}{2}$ is trivial since we can simply guess the (binary) output of f. Hence, this is really the minimal condition for A to be "non-trivial."

Definition 7 (Function Computation (Relaxed)) *A randomized algorithm A is said to $\epsilon(\cdot)$-compute a function $f : \{0,1\}^* \to \{0,1\}$ if $\forall x \in \{0,1\}^*$, A on input x outputs $f(x)$ with probability $\epsilon(|x|)$.*

Theorem 1 (Amplification) *Let A be a randomized algorithm that $\epsilon(\cdot)$-computes $f : \{0,1\}^* \to \{0,1\}$ for $\epsilon(n) = \frac{1}{2} + \frac{1}{q(n)}$ where $q(\cdot)$ is a polynomial. Then, there exists an algorithm A' that $\epsilon'(\cdot)$-computes f, where $\epsilon'(n) = 1 - \frac{1}{2^{|x|}}$.*

The above theorem states that we can *amplify* the success probability of A from slightly more than one half to nearly 1. The main idea is to run A multiple times (on the same input), using fresh random tape each time, and then take the majority vote.

Try to implement this idea for the case where $q(\cdot)$ is simply a constant. For example, say $\epsilon = \frac{3}{4}$. Then extend the proof to the general case. The formal proof uses Chernoff Bound.

3 Adversaries

How should we model an adversary? We don't want to place too many assumptions or limitations on adversaries. In particular, we only want to assume that adversaries are "efficient," but make no other assumption regarding their behavior.

To capture efficient adversaries, we can model them as PPT algorithms. In particular, we will consider *non uniform* PPT algorithms, abbreviated as n.u. PPT.

Definition 8 (Adversary) *A non-uniform PPT algorithm A (aka adversary) is a sequence of probabilistic machines* $\{A_1, A_2, A_3, \dots\}$ *for which* $\exists$ *polynomial* $p(\cdot)$ *s.t* $\forall$ *input* x, $A_{|x|}$ *on input* x *halts in time* $p(|x|)$*. In other words, an adversary has a dedicated Turing Machine for every input length.*

Note that the weaker definition that models an adversary as a (uniform) PPT algorithm does not capture adversaries that may be able to build a dedicated machine that breaks a cryptosystem on a particular input length (but fails otherwise). The above definition that allows an adversary to be a non-uniform PPT captures this case, and is therefore, stronger.

4 One Way Functions

We now consider the notion of one-way functions. Intuitively, a function f is *one way* if it is possible to compute $f(x)$ efficiently but "hard" to recover x given $f(x)$.

Definition 9 (One Way Function (Informal Attempt)) *A function* f *is one way if:*

- *It is "easy" to compute* $f(x)$ *given* x*.*
- *"Difficult" to compute* $x \in f^{-1}(y)$ *given* $y = f(x)$*.*

Remark 3 *It may be possible to partially recover* x*. However, it should be hard to fully recover* $x \in f^{-1}(y)$*. In other words, for any adversary A, the probability that A can compute* x *given* $f(x)$ *is "small".*

In the next class, we will define one-way functions formally.