Lecture 18: Introduction to "Security against Computationally Bounded Adversaries"

Outline

- Till the previous lecture, the security of our constructions held against adversaries even if they have unbounded computational power
  - For example, suppose a secret s is shared among 5 parties using Shamir's secret sharing algorithm such that any set of 3 parties can reconstruct the secret, and the secret is hidden from the collusion of any 2 parties
  - This security holds even if the parties has unbounded computational power!
  - Security guarantees against adversaries with unbounded computational power is ideal, but most cryptography is impossible in this setting
  - So, we relax the notion of security. We ensure security only against adversaries that are <u>efficient</u>

Efficient Algorithm

- Intuitively, an algorithm is efficient if the running time of the algorithm is upper-bounded by a polynomial in its input length
  - For example, the algorithm $\mathrm{Multiply}(x, y)$ takes as input two n-bit inputs x and y and outputs the binary representation of the product of the two numbers x and y. The length of the input of this algorithm is $\left|(x, y)\right| = 2n$. Note that the number x is exponentially larger than the "length of x." For instance, the number 100 needs only 7 bits for binary representation
  - The algorithm $\mathrm{Prime}(x)$ takes as input an n-bit input x and tests whether it is a prime or not. And efficient algorithm to test primality will have running time polynomial in n
  - $\mathrm{GCD}(x, y)$ is the algorithm that takes n-bit numbers x and y, and outputs the binary representation of their greatest common divisor. And efficient algorithm to compute the GCD of integers will have running time at most a polynomial in n

Example I

- Let us consider the example of multiplying two n-bit numbers
- Consider the following code

Multiply-v1 (x, y): 1. Let $r = 0$ 2. For $i \in [1, \ldots, y]$: $r+ = x$ 3. Return r

- This is a <u>correct</u> algorithm to multiply the two numbers x and y. But its running time is proportional to y, which can be exponential in n. So, this algorithm is <u>not efficient</u>

Example II

- Let us consider another code of multiplying two n-bit numbers
- Consider the following code

> Multiply-v2 (x, y):
>
> 1. Let M be the table that stores the answer $x \times y$ at the matrix entry (x, y)
> 2. Perform binary search (or direct memory addressing) to find the entry $M(x, y)$ and output this entry

- Binary search takes time linear in n. But the length of the overall code is 2^{2n}. This is also considered inefficient

Example III

- Consider the following code to multiply two n-bit numbers

Multiply-v3 (x, y):

1. Let $x_0 x_1 \ldots x_{n-1}$ be the binary representation of x
2. Let $y_0 y_1 \ldots y_{n-1}$ be the binary representation of y
3. Let $c = 0$ (carry bit)
4. For $i \in \{0, \ldots, n-1\}$:
    - $t = x_i + y_i + c$ (addition over integers)
    - If $t \geqslant 2$ then set $c = 1$, else $c = 0$
    - Let $z_i = (t\%2)$
5. Let $z_n = c$
6. Return $z_0 z_1 \ldots z_{n-1} z_n$

- The length of this code is linear in n and its running time is also linear in n
- This is an efficient algorithm for addition

Another Example I

- Suppose we want to test whether an n-bit input is a prime number or not

Is_Prime (x):

- For $i \in \{2, \ldots, \lfloor\sqrt{x}\rfloor\}$: If i divides x then return false
- Return true

- This algorithm runs in time proportional to $\sqrt{x}$, which is exponential in n. This is not an efficient algorithm for primality testing!

Another Example II

- Until (roughly) 15 years ago, we only knew a probabilistic algorithm that was efficient
- It was a very big open problem to design a deterministic efficient algorithm for primality testing
- Finally, Agrawal-Kayal-Saxena (AKS) provided the first deterministic primality testing algorithm

Factoring

- Consider the task of finding a divisor of a $2n$-bit number x
- When $x = pq$, where p and q are n-bit prime numbers, we believe that there is no efficient algorithm for this task
- Note that this is a believe and not proven!
- Note that, it is easy to find a divisor when x is even. It may also be easy to find divisors of x when x has small prime factors. But when x is the product of two n-bit prime numbers, then we believe that finding a divisor of x is hard.

Food for Thought

- Write down efficient algorithms for the following tasks
  - Perform division of x by y (output both the quotient and the remainder)
  - Finding the GCD of x and y, two n-bit integers
  - Multiply two polynomial $p(X)$ and $q(X)$ that are of degree n and have binary coefficients
  - Multiply two $n \times n$ matrices with field entries
  - Find g^x, where g is a generator of a group
- Read about the Fast Fourier Transform

Why do Hard Problems Exist?

- If the complexity class P is equal to the complexity class NP, then there are no hard problems
- For example, suppose given a 3-SAT formula ϕ over n variables we can efficiently determine whether it has a satisfiable solution or not. If this is the case then $\mathrm{P} = \mathrm{NP}$. And in this case, there will be no hard problems
- So, cryptographers rely on $\mathrm{P} \neq \mathrm{NP}$ and additional assumptions ...

One-way Functions I

- Let $f\colon \{0,1\}^n \to \{0,1\}^n$ be a function that can be computed in polynomial time (i.e., polynomial in n)
- Consider any efficient adversary $\mathcal{A}$
- Define the following experiment
  1. Sample $x \xleftarrow{\$} \{0,1\}^n$
  2. Compute $y = f(x)$
  3. Give y to the adversary $\mathcal{A}$
  4. Obtain its reply $x' = \mathcal{A}(x)$
  5. Let $z = (f(x') == y)$
- We want the probability

$$\mathbb{P}\left[z = \text{true} \colon x \xleftarrow{\$} \{0,1\}^n, y = f(x), z = \big(\mathcal{A}(y) == y\big)\right] \leqslant 2^{-cn}$$

One-way Functions II

Explanation of the definition

- So, <u>one-way functions</u> are
  - Easy to evaluate, but
  - Hard to invert
- The variable z takes value $\{\text{true}, \text{false}\}$. It is true if and only if the adversary $\mathcal{A}$ produces a pre-image of y
- Note: we insist that the adversary has to produce <u>any</u> pre-image of y. It need not necessarily produce x
  - For example, a function $f(x) = 0$ for all $x \in \{0,1\}^n$ is not a one-way function. Because, consider the adversary that outputs $\mathcal{A}(y) = 0^n$. We always have $f(0^n) = f(x)$. Hence, the probability of $z = \text{true}$ is 1!

One-way Functions III

- If P = NP then one-way functions cannot exist! (We can efficiently invert any function. Think: Why?)

One-way Functions IV

- A weak one-way function has

$$\mathbb{P}\left[z = \mathsf{true} \colon x \xleftarrow{\$} \{0,1\}^n, y = f(x), z = \left(\mathcal{A}(y) == y\right)\right] \leqslant 1 - \frac{1}{\mathrm{poly}(n)}$$

- If weak one-way functions exist then one-way functions also exist. That is, given any weak one-way function we can construct a one-way function.

Next Lecture

- We shall consider candidate constructions of one-way and weak one-way functions (we believe that these functions are one-way functions)