Lecture Notes 1

1 Course Overview

General: See the syllabus for course overview, textbooks, requirements, and policies.

Prerequisites: This course assumes a basic knowledge of cryptography as covered by CPSC 467a. It is based on probability theory and complexity theory. I will try to cover what is needed from each of these topics, but a background in probability theory as covered by STAT 238a or STAT 241a and a background in computational complexity as covered by CPSC 468a will certainly help. In particular, the first few chapters of the textbook used in CPSC 468a, Complexity Theory: A Modern Approach, by Sanjeev Arora and Boaz Barak are a useful reference for basic definitions in complexity theory.

Major topics: Some of the major topics to be covered include

- One-way functions and hard core predicates
- Pseudorandom generators
- Zero knowledge proof systems
- Encryption
- Signatures
- Secure computation
- Multiparty protocols

Course format: I will conduct the first part of the course as a regular lecture course covering material from the textbook. The remainder of the course will be conducted in a seminar style, with each class devoted to a particular paper or topic. Each student will be expected to act as discussion leader for one or two papers during the term. The topics of the selected papers are not narrowly restricted to these foundational issues but can include papers of special current interest, such as the recent paper by Sotirov et al., MD5 considered harmful today, which shows how to exploit a weakness in MD5.

2 Some Background from Probability Theory

This material is contained in section 1.2 of the textbook, so I only state some of the major theorems here.

Let X be a discrete random variable. The *expected value* of X is given by

$$E(X) = \sum_x x \cdot \Pr[X = x].$$

Another name for the expected value is the *mean*.

Theorem 1 (Markov inequality) *Let X be a non-negative random variable.*

$$\Pr[X \geq v] \leq \frac{E(X)}{v}.$$

The intuition here is that X can't be much larger than the mean very often without causing the mean itself to rise. For example, if $\Pr[X \geq 20]$ is more than 0.1, the expected value would have to be bigger than 2. Hence, if we know that $E(X) = 2$, we can conclude that $\Pr[X \geq 20] \leq 0.1$, as asserted by the theorem.

The variance of X is given by

$$\mathrm{Var}(X) = E\left((X - E(X))^2\right).$$

It gives a measure of how much X is likely to deviate from the mean.

Theorem 2 *Chebyshev's inequality*

$$\Pr[|X - E(X)| \geq \delta] \leq \frac{\mathrm{Var}(X)}{\delta^2}.$$

Recall that $E(X)$ is actually a constant, the mean of X, even though it looks like a random variable.

Corollary 3 (Pairwise sampling) *Let $X_1, \ldots, X_n$ be pairwise independent random variables, all with mean μ and variance σ^2. Then for all $\varepsilon > 0$, we have*

$$\Pr\left[\left|\frac{\sum X_i}{n} - \mu\right| \geq \varepsilon\right] \leq \frac{\sigma^2}{\varepsilon^2 n}.$$

Theorem 4 (Chernoff bound) *Let $p \leq \frac{1}{2}$ and let $X_1, \ldots, X_n$ be independent 0-1 valued random variables such that $\Pr[X_i = 1] = p$ for each i. Then for all $\varepsilon, 0 \leq \varepsilon \leq p(1-p)$, we have*

$$\Pr\left[\left|\frac{\sum X_i}{n} - p\right| \geq \varepsilon\right] \leq 2e^{-\frac{\varepsilon^2 n}{2p(1-p)}}.$$

The Hoefding inequality generalizes the Chernoff bound to general independent identically distributed random variables. See the text for further details.

3 Turing Machines

In order to talk about computational difficulty, we need a well-defined model of computation and a resource measure such as amount of time to carry out a computation. It doesn't much matter what model we use as long as it is both sufficiently powerful and reasonable. Sufficiently powerful means that it has enough memory to carry out arbitrarily large computations. Reasonable means that it operates "mechanically" according to a finite "program". Most any modern computer or programming language meets the requirement of reasonableness. To be sufficiently powerful means that enough memory must be available to carry out any computation. While nobody can actually build a machine with infinite storage capabilities, we can certainly imagine a C program, for example, in which `malloc()` never fails when more memory is requested.

For definiteness, it is conventional to take a Turing machine as the specific computational model. Because a Turing machine can simulate any other reasonable model of computation with reasonable efficiency, results about Turing machines translate into similar results about your favorite model and are thus "model invariant" in a pretty strong technical sense.

3.1 The model

A Turing machine consists of a finite memory and an infinite tape, divided into discrete cells. Each cell can store one of a finite set Σ of *tape symbols*. We assume the special symbol blank symbol $\sqcup$ is in Σ and that all but finitely many cells of the tape initially contain $\sqcup$.

A read/write head always resides on some tape cell. Primitive operations let the machine read the symbol stored in the cell under scan or replace that symbol with a new one. The machine also has operations that move the head one cell to the left or one cell to the right of its current position.

The operations of the machine are controlled by a finite program. Depending on the contents of its finite memory, the machine can take a *step* that performs a tape operation and changes the contents of its memory. A memory configuration that does not allow further steps to be taken is said to be *halting*.

A *halting computation* consists of a finite sequence of steps from an initial configuration to a halting configuration. The number of steps in the computation is the *running time* for that machine on that initial configuration.

3.2 Computation of functions and predicates

We use Turing machine to perform computations over a domain of interest. Two such domains are the set of finite binary strings $\{0,1\}^*$ and the set of natural numbers. We typically want to compute functions and predicates over those domains. To this end, we need conventions for presenting the arguments of the function or predicate to the machine and for interpreting the output.

Many encodings are possible. One simple one involves a 4-symbol tape alphabet $\Sigma = \{\sqcup, 0, 1, *\}$. To present k binary strings $x_1, \ldots, x_k$ to the machine, we initialize the non-blank portion of the tape to $*\,x_1 * x_2 * \ldots * x_k *$ and place the read/write head over the leftmost $*$. To present k natural numbers, we represent each number as a binary string and present the resulting strings as just described.

The machine produces an *output* string y only if the machine eventually halts, the non-blank portion of the tape consists of the string $*\,y\,*$, and the read/write head resides on the leftmost $*$. If the machine does not halt, or if the tape is not as just described when it halts, the output is undefined.

A Turing machine computes a k-ary function f if it halts on all inputs $x_1, \ldots, x_k$ encoded as described above and gives output $y = f(x_1, \ldots, x_k)$. To compute predicates and relations, we regard them as functions with outputs in $\{0,1\}$, where 0 represents false and 1 represents true.

3.3 Time complexity

Let M be a Turing machine that computes $f()$. Its running time $\text{Time}_M(\cdot)$ is the number of steps until the machine halts, taken as a function of the k inputs.

The actual running time function contains more information than we are generally interested in. We typically are satisfied in providing upper and lower bounds on the time as a function of just the total length n of the inputs. For example, we say that M runs in time $T(n) = n^2 + 5$ if it is the case that $\text{Time}_M(x) \leq T(|x|)$, where $|x|$ is the length of the string x.

We also often use the "big-oh" notation to disregard constant factors and a finite set of anomolous values. For functions f and g, we write $f = O(g)$ (or equivalently, $f(n) = O(g)$) to mean that $f(n) \leq cg(n)$ for some constant c and all sufficiently large n. For example, $3n + 100 = O(n)$ since $3n + 100 \leq cn$ for $c = 4$ and all $n \geq 100$.

3.4 Complexity class $\mathcal{P}$

A *language* is a set of binary strings. A language L can be *decided* in time $T(n)$ if there is a Turing machine that computes the membership predicate for L and runs in time $T(n)$. That is, for any string x, $M(x)$ halts within $O(T(|x|))$ steps and outputs 1 if $x \in L$ and 0 if $x \notin L$.

The complexity class $\mathcal{P}$ consists of the set of all languages that can be computed in time bounded by some polynomial. Thus, we can write

$$\mathcal{P} = \bigcup_{k}^{\infty} \{L \subseteq \{0,1\}^* \mid \text{ some machine } M \text{ decides } L \text{ in time } O(n^k)\}.$$

We take computability in polynomial time as our notion of "feasible computation". Thus, $\mathcal{P}$ consists of those languages for which membership can be feasibly tested.

3.5 Complexity class $\mathcal{NP}$

The complexity class $\mathcal{NP}$ is the set of languages for which membership can be proved in a feasible amount of time. We say that a language L is in $\mathcal{NP}$ iff there is a relation $R_L(x, y)$ and polynomials $p(n)$ and $q(n)$ with the following properties:

1. $R_L(x, y)$ can be computed in time $p(n)$.
2. $L = \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^* (|y| \leq q(|x|) \wedge R_L(x, y))\}$.

We say that y is a *witness to x's membership in L* iff $|y| \leq q(|x|)$ and $R_L(x, y)$ is true. Thus, $x \in L$ iff there exists a witness to its membership in L. We can think of y as a "proof" that $x \in L$ and the machine computing R_L as a verifier for the correctness of such proofs.

In summary, $\mathcal{P}$ consists of those languages for which membership can be tested in a feasible amount of time, and $\mathcal{NP}$ consists of those language for which polynomial length proofs of membership can be verified in a feasible amount of time.

To test your understanding make sure you can prove the following:

Theorem 5 (easy) $\mathcal{P} \subseteq \mathcal{NP}$.