COM S 687 Introduction to Cryptography August 31, 2006

Lecture 3: One-way functions

Instructor: Rafael Pass *Scribe: Lucja Kot*

1 Review

We have seen that no encryption scheme can be perfectly secure if the keys it uses are shorter than the messages, even if the length difference is just one bit. We proved:

Theorem 1 *Let $E = (\mathcal{M}, \mathcal{K}, Gen, Enc, Dec)$ be a* deterministic *private-key encryption scheme where $\mathcal{M} = \{0,1\}^n$ and $\mathcal{K} = \{0,1\}^{n-1}$. Then E is not more than $1/2$-statistically secret.*

This was proved by demonstrating the existence of m_0, $m_1 \in \mathcal{M}$, such that if we let $T = \{c \mid \exists k \; Enc_k(m_0) = c\}$,

$$\mathsf{Pr}\left[Enc(m_0) \in T\right] - \mathsf{Pr}\left[Enc(m_1) \in T\right] \geq 1/2.$$

We saw an attack that exploits the above vulnerability. Suppose the adversary (Eve) receives a ciphertext c. Eve knows that c is an encryption of either m_0 or m_1, and these two messages were sent with equal probability. She can compute as follows:

1. Compute the set $M' = \{m \in \mathcal{M} \mid \exists \; k \in \mathcal{K} \; Dec_k(c) = m\}$
2. If $m_0 \in M'$, output m_0, otherwise output m_1.

We argue that this algorithm will output the message that was sent with probability $\geq \frac{3}{4}$. Consider first the case where m_0 was sent. Then the attack algorithm will output m_0 with probability 1. On the other hand, suppose m_1 was sent. Then, by the above Theorem, the algorithm will output m_1 with probability $\geq \frac{1}{2}$. As we assumed m_0 and m_1 have an equal probability of being sent, we see that Eve can indeed find the message with probability $\geq \frac{1}{2} \times 1 + \frac{1}{2} \times \frac{1}{2} = \frac{3}{4}$.

We closed last time by noting that the above attack requires exponential time, because the computation of M' requires computing a decryption for each $k \in \mathcal{K}$. Consequently the attack, while worrying, is not in fact computationally feasible – particularly when n is a large enough number. This motivates our introduction of an adversary model where computation time is a bounded resource.

2 Computational Hardness and Efficient Adversaries

2.1 Deterministic Computation

We start by formalizing what if means for an algorithm to compute a function.

Definition 1 (Algorithm) *An* algorithm *is a (deterministic) Turing machine whose input and output are strings over some alphabet Σ. We usually have $\Sigma = \{0,1\}$.*

Definition 2 (Running-time of Algorithms) *$\mathcal{A}$ runs in time $T(n)$ if for all $x \in B^*$, $\mathcal{A}(x)$ halts within $T(|x|)$ steps. $\mathcal{A}$ runs in* polynomial time *(or is an* efficient *algorithm) if $\exists$ c such that $\mathcal{A}$ runs in time $T(n) = n^c$.*

Definition 3 (Deterministic Computation) *Algorithm $\mathcal{A}$ is said to* compute *a function $f : \{0,1\}^* \to \{0,1\}^*$ if $\mathcal{A}$, on input x, outputs $f(x)$, for all $x \in B^*$.*

Remark: It is possible to argue with the choice of polynomial-time as a cutoff for "efficiency", and indeed if the polynomial involved is large, computation may not be efficient in practice. There are, however, strong arguments to use the polynomial-time definition of efficiency:

1. This definition is independent of the representation of the algorithm (whether it is given as a Turing machine, a C program, etc.) as converting from one representation to another only affects the running time by a polynomial factor.
2. This definition is also closed under composition, which is desirable as it simplifies reasoning.
3. "Usually", polynomial time algorithms do turn out to be efficient ('polynomial" almost always means "cubic time or better")
4. Most "natural" functions that are not polynomial-time computable require *much* more time to compute, so the separation we propose appears to have solid natural motivation.

Remark: Note that our treatment of computation is an *asymptotic* one. In practice, actual running time needs to be considered carefully, as do other "hidden" factors such as the size of the description of $\mathcal{A}$. Thus, we will need to instantiate our formulae with numerical values that make sense in practice.

2.1.1 Some computationally "easy" and "hard" problem

We know many functions computable by efficient algorithms. Among them are the following.

1. $DIV(X, Y) = (q, r)$ such that $x = yq + r$, computable by any standard integer division algorithm.
2. $GCD(X, Y)$, the largest Z such that $Z \mid X$ and $Z \mid Y$, computable by the Euclidean algorithm.
3. $MODEXP(X, Y, Z) = X^Y \ mod \ Z$, computable by a standard algorithm involving binary exponentiation in time polynomial in the logarithm of the input.

There are also functions which are known or believed to be hard.

1. Given a description of a Turing machine M, determine whether or not M halts – this is an example of an uncomputable problem.
2. There exists functions $f : \{0,1\}^* \to \{0,1\}$ that are computable, but are not computable in polynomial time (their existence is guaranteed by the Time Hierarchy Theorem in Complexity theory)
3. SAT, the problem of determining whether a Boolean formula has a satisfying assignment. SAT is *conjectured* not to be polynomial-time computable – this is the very famous $P \neq NP$ conjecture.

2.2 Randomized Computation

In practice we need to consider adversaries that also have access to some source of randomness. We extend the above definitions to capture also such adversaries.

Definition 4 (Randomized Algorithms - Informal) *A* randomized algorithm *is a Turing machine equipped with an extra tape that where each bit of the tape has been uniformly and independently chosen.*

Equivalently, a randomized algorithm is a Turing Machine that has access to a "magic" randomization box that output a truly random bit at demand.

To define efficiency we must clarify the concept of *running time* for a randomized algorithm. There is a subtlety that arises here, as the actual run time may depend on the bit-string obtained from the random tape. We take a conservative approach and define the running time as the upper bound over all possible random sequences.

Definition 5 (Running-time of Randomized Algorithms) *A randomized Turing machine $\mathcal{A}$ runs in time $T(n)$ if for all $x \in B^*$, $\mathcal{A}(x)$ halts within $T(|x|)$ steps (independent of the content of $\mathcal{A}$'s random tape). $\mathcal{A}$ runs in* polynomial time *(or is an* efficient *randomized algorithm) if $\exists$ c such that $\mathcal{A}$ runs in time $T(n) = n^c$.*

We extend our definition of computation to randomized algorithm.

Definition 6 *Algorithm $\mathcal{A}$ is said to* compute *a function $f : \{0,1\}^* \to \{0,1\}^*$ if $\mathcal{A}$, on input x, outputs $f(x)$ with probability $\geq \frac{2}{3}$ for all $x \in B^*$.*

At first sight the bound $\frac{2}{3}$ might seem arbitrary. However, it can be shown (as in homework 1) that the same class of functions will be computable by efficient randomized algorithms even if replacing the bound with either $\frac{1}{2} + \frac{1}{poly(|x|)}$ or $1 - 2^{-|x|}$. In other words, given a polynomial-time randomized algorithm $\mathcal{A}$ that computes a function with probability $\frac{1}{2} + \frac{1}{poly(n)}$, it is possible to obtain another polynomial-time randomized machine $\mathcal{A}'$ that computes the function with probability $1 - 2^{-n}$.($\mathcal{A}'$ simply takes multiple runs of $\mathcal{A}$ and finally outputs the most frequent output of $\mathcal{A}$. The Chernoff bound can then be used to analyze the probability with which such a "majority" rule works.)

Efficient Adversaries. Polynomial-time randomized algorithms will be the principal model of efficient computation considered in this course. In the sequel, we will employ the terms polynomial-time randomized algorithm, *probabilistic polynomial-time Turing machine* (*p.p.t*, or PPT), *efficient randomized algorithm*, or simple *feasible algorithm* interchangeably.

It is worthwhile to revisit the three above mentioned "hard" problems with respect to randomized computation.

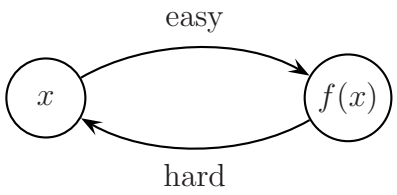
1. Since the halting problem is not computable, it is also not computable by randomized algorithm.
2. However, it is unknown whether there exists functions $f : \{0,1\}^* \to \{0,1\}$ that are computable by say exponential-time randomized algorithms, but not computable by polynomial randomized algorithms. This is a very interesting open problem in Complexity Theory (i.e., establishing a, so called, *probabilistic time hierarchy theorem*).
3. The hardness of SAT for efficient randomized algorithms is another famous conjecture – $NP \neq BPP$.

3 One-Way Functions

Computationally hard functions are essential, but not (to our knowledge) sufficient, to produce encryption schemes. It turns out that we require functions with specific properties, hardness being one of them.

At a high level, there are two basic desiderata for any encryption scheme:

- it must be feasible to generate c given m and k, but
- it must be hard to recover m and k given c.



This suggests that we require functions that are easy to compute but hard to invert – *one-way functions*. Indeed, these turn out to be the most basic building block in cryptography.

There are several ways that the notion of one-wayness can be defined formally. We start with a definition that formalizes our intuition in the simplest way.

Definition 7 (Worst-case One-way Function) *A function* $f : \{0,1\}^* \rightarrow \{0,1\}$ *is (worst-case)* one-way *if:*

1. *there exists a p.p.t (probabilistic polynomial time Turing machine)* $\mathcal{C}$ *such that* $\mathcal{C}(x) = f(x)$*, and*
2. *there is no p.p.t algorithm* $\mathcal{A}$ *such that* $\forall x \ Pr\ [\mathcal{A}(f(x)) \in f^{-1}(f(x))] \geq \frac{2}{3}$

We will see that assuming $SAT \notin BPP$, one-way functions according to the above definition must exist (in fact, you will show that these two assumptions are equivalent). Note, however, that this definition allows for certain pathological functions – those where inverting the function for *most* x values is easy, as long as every machine fails to invert f(x) for infinitely many x's. It is an open question whether such functions can still be used for good encryption schemes. This observation motivates us to refine our requirements. We want functions where for a randomly chosen x, the probability that we are able to invert the function is very small. With this new definition in mind, we begin by formalizing the notion of *very small*.

Definition 8 *A function $\varepsilon(n)$ is* negligible *if for every c, there exists some k_0 such that $\epsilon(k) \leq \frac{1}{k^c}$ for all $k_0 < k$. Intuitively, a negligible function is asymptotically smaller than the inverse of any fixed polynomial.*

We are now ready to present a more satisfactory definition of a one-way function.

Definition 9 (Strong One-way Function) *A function $f : \{0,1\}^* \to \{0,1\}$ is* (strongly) one-way *if:*

1. *there exists a p.p.t algorithm $\mathcal{C}$ such that $\mathcal{C}(x) = f(x)$, and*
2. *for every p.p.t algorithm $\mathcal{A}$ there exists a negligible function $\epsilon(k)$ such that $\forall k$, we have:*

$$Pr\ [x \leftarrow \{0,1\}^k,\ y = f(x),\ \mathcal{A}(1^k, y) = x' \ :\ f(x') = y] \ \leq\ \epsilon(k)$$

Remark:

1. The algorithm $\mathcal{A}$ receives the additional input of 1^k; this is to allow $\mathcal{A}$ to take time polynomial in $|x|$, even if the function f should be substantially length-shrinking. In essence, we are ruling out some pathological cases where functions might be considered one-way because writing down the output of the inversion algorithm violates its time bound.
2. As before, we must keep in mind that the above definition is *asymptotic*. To define one-way functions with concrete security, we would instead use explicit parameters that can be instantiated as desired. In such a treatment we say that a function is (t, s, ϵ)-one-way, if no $\mathcal{A}$ of size s with running time $\leq\ t$ will succeed with probability better than ϵ.