CS 810: Complexity Theory 4/11/2007

Lecture 26: Cryptographic Primitives

Instructor: Dieter van Melkebeek Scribe: Chi Man Liu

In this lecture, we give an overview of the complexity-theoretic aspects of cryptography. We focus on confidentiality and look at one cryptographic primitive: the one-way function. Existence of one-way functions is still an open question. We give some examples of candidate one-way functions, and relate the existence of one-way functions to some other complexity-theoretic hypotheses, in particular $\mathrm{NP} \not\subseteq \mathrm{P/poly}$.

1 Overview

Cryptography is the study of secure interaction between two or more parties. In the context of complexity theory, "parties" usually refer to efficient machines, and "interaction" refers to digital communication. There are several common interpretations for "secure".

1. *Confidentiality*: transforming data into other forms in order to prevent unauthorized parties from knowing the content by looking at the communication.
2. *Integrity*: ensuring that the received message comes from the right party, and its content has not been changed.
3. *Authenticity*: verifying the identity of trusted parties.

Confidentiality, integrity and authenticity are the traditional goals of cryptography. Other goals, such as privacy, have also been studied. In this lecture, we focus on confidentiality — hiding information from others.

2 Confidentiality

We consider a very simple model. Suppose that Alice wants to send a message M to Bob over some channel. An eavesdropper, Eve, tries to figure out what Alice has sent to Bob by intercepting the communication between them. Therefore, to secure the message against Eve, Alice has to massage M into something else before sending it to Bob. After receiving the message, Bob recovers the original content of M by some operation. In order for this scheme to work, either Alice or Bob (or both) must know something more than Eve does; otherwise, Eve would have simulated Bob's actions to recover M.

Formally, Alice runs an encryption algorithm E with some encryption key K_E on the plaintext M. The algorithm outputs a ciphertext C, which is sent to Bob over the channel. At Bob's side, a decryption algorithm D with decryption key K_D is run on the ciphertext C. We expect the decryption algorithm to output M. We need to pick E, D, K_E and K_D such that Eve is unable to recover M by just looking at C.

We look at two different settings. The first one is *private key systems* (also known as symmetric key systems) where $K_E = K_D$. The second one is *public key systems* (also known as asymmetric key systems) where $K_E \neq K_D$. Moreover, we require that K_E can be easily computed from K_D but not the other way round.

2.1 Private Key Systems

In a private key system, there is only one key K, which is used both for encryption and decryption. Note that although only one key is used, the encryption algorithm may be different from the decryption algorithm.

A very simple private key system is the *one-time pad*. The encryption algorithm outputs $E_K(M) = M \oplus K$. The decryption algorithm outputs $D_K(C) = C \oplus K$. We pick K uniform at random from $\{0,1\}^n$, where n is the length of the message. No matter how much computational resources the eavesdropper has, she cannot correctly determine the plaintext M with a probability higher than 2^{-n}. Information theoretically, this is the best we can hope for. In other words, the one-time pad system has perfect security.

One drawback to the one-pad time system is that the key needs to have the same length as the message. If we want to send a message of size n, we have to generate n random bits. We can relax the notion of security from information-theoretic to complexity-theoretic. Instead of requiring the system to be secure against eavesdroppers with unlimited computational resources, we only require it to be secure against computationally bounded adversaries, e.g. polynomial-time machines. We achieve this by using pseudorandom generators with short seed length that fool computationally bounded adversaries. With a such PRG, we can first pick a short random string and then apply the PRG to get a long pseudorandom string K which is used as the key.

The PRGs we consider here are different from the PRGs we used for derandomization. Instead, we look at *cryptographic PRGs* (CPRGs). There are a few differences between CPRGs and PRGs for derandomization.

- *Efficiency.* When we discussed time-bounded derandomization, we only require PRGs to be computable in linear exponential time. However, we expect the generation of encryption keys to be efficient, hence linear exponential time wouldn't work for CPRGs. We want the running time of CPRGs to be polynomial.

- *Strength.* When derandomizing certain complexity classes, we know the "adversary" (the target to be fooled), e.g. circuits of quadratic size for the case of BPP. In our cryptographic setting, the adversary we are trying to fool is a polynomial-time computing device. However, we do not know the degree of that polynomial. The CPRG runs in time $O(n^c)$ for some fixed constant c, yet it has to fool any $O(n^d)$-time adversary with arbitrarily large constant d. This implies that our "hardness versus randomness" technique applied to constructing quick PRGs for circuits wouldn't work for CPRGs.In later discussions, we focus on CPRGs against nonuniform models.

- *Seed length or stretch.* In derandomization, we want the seed length to be $O(\log r)$, i.e. the stretch is at least linear exponential. In cryptography, we only require the stretch to be polynomial.

We will talk more about CPRGs in the next lecture.

Other examples of private key systems include the Data Encryption Standard (DES) and the Advanced Encryption Standard (AES). These systems have been shown to be secure under some specific complexity-theoretic hypotheses.

2.2 Public Key Systems

In a public key system, the encryption key K_E and the decryption key K_D are different in general. K_E can be computed by K_D efficiently, but not the other way round. Here is how it works. First, Bob picks K_D randomly and out of it he computes K_E. Then, he makes K_E public to everyone he expects to receive messages from, while keeping K_D private. Now, if someone wants to send Bob a message, she has to encrypt the message with the *public key* K_E and send the ciphertext over the channel. After receiving the ciphertext, Bob uses his *private key* K_D to decrypt it back into plaintext message. A widely used public key system is the RSA encryption system.

One advantage of public key systems over private key systems is that when m parties want to communicate with each other, only m pairs of keys are needed using a public key system, whereas $O(m^2)$ are needed in a private key system. However, many private key systems are still being used. One reason is that most public key systems are more computationally intensive than private key systems. Besides, known algorithms for breaking public key systems are more efficient than those for private key systems, although they all run in exponential time. As a result, the key size of a public key system tends to be larger than that of a private key system.

3 One-Way Functions

The encryption systems in the previous section become vulnerable if $P = NP$. However, $\mathrm{P} \neq \mathrm{NP}$ does not guarantee good security. Practically, we want it to be case that $\mathrm{NP} \not\subseteq \mathrm{BPP}$, or even $\mathrm{NP} \not\subseteq \mathrm{P/poly}$. $\mathrm{NP} \not\subseteq \mathrm{BPP}$ rules out the possibility of having efficient randomized algorithms for breaking the above systems. $\mathrm{NP} \not\subseteq \mathrm{P/poly}$ states that NP does not have polynomial-size circuits. These encryption systems would become insecure if NP has polynomial-size circuits, for one can compute polynomial-size circuits for certain input sizes (possibly using a huge amount of resources), and use them to break the systems efficiently on those input sizes. Yet, these assumptions are still not enough to ensure confidentiality. In this section, we look at one-way functions, which capture the notion of hardness in computing the decryption key given the encryption key. Intuitively, a one-way function is a function which is easily computable in one direction, but hard to invert on average.

3.1 Definition

There are two issues in encryption systems that do not seem to be solved by just assuming $\mathrm{NP} \not\subseteq \mathrm{P/poly}$.

- We look at the *average-case hardness* of a function instead of worst-case hardness. We want the function to be hard to invert for not just a few inputs but most.
- We require the existence of an efficient procedure to generate "solved" hard instances.

Definition 1. *Given a function $f : \{0,1\}^* \to \{0,1\}^*$, the* hardness of inverting f on input size n*, denoted $HI_f(n)$, is the largest integer s such that no circuit C of size at most s succeeds in inverting f on at least a $1/s$ fraction of the inputs of size n.*

Note that $s = HI_f(n)$ is at most exponential in n, since any Boolean function with n inputs has circuits of size $2^{O(n)}$. As s gets smaller, the circuit C becomes more computationally restrictive, yet it is required to successfully invert a larger fraction of inputs. Thus $HI_f(n)$ serves as a measure of average-case hardness of inverting f on input size n.

We need a more precise definition for the statement "a circuit C succeeds in inverting a function f on at least a $1/s$ fraction of the inputs of size n". For a permutation (a one-to-one onto function) f, the preimage of an output is always unique. The statement can be restated as

$$\Pr_{|x|=n}[C(f(x)) = x] \geq \frac{1}{s}.$$

For a general function f, an output may have multiple preimages. We only require C to compute one of the preimages, that means

$$\Pr_{|x|=n}[f(C(f(x))) = f(x)] \geq \frac{1}{s}.$$

We now give the definition of one-way functions.

Definition 2. *A function $f : \{0,1\}^* \to \{0,1\}^*$ is a* one-way function *if f is polynomial-time computable and $HI_f(n) \geq n^{\omega(1)}$.*

In the uniform setting, circuits are replaced by any polynomial-time randomized algorithm. Intuitively, a function f is hard to invert if for any polynomial-time randomized algorithm A, A fails to invert f on a significant fraction of the inputs as the input size gets larger.

3.2 Candidate One-Way Functions

We give a few candidates for one-way functions. Note that they are candidates only — one-way functions do not exist if P = NP.

- *Multiplication of primes:*
$$f(p, q) = pq,$$
where p and q are primes. Factorization of integers is believed to be a hard problem.

- *Squaring modulo pq:*
$$f(p, q, x) = (pq, x^2 \bmod pq),$$
where p, q are primes and x is an integer. It can be shown that inverting this function is equivalent to factoring in terms of average-case complexity. As an aside, finding square roots modulo p is a very easy problem. That's why we need to use modulo pq.

- *Exponent modulo p:*
$$f(p, g, x) = (p, g, g^x \bmod p),$$
where p is a prime, g is a generator of the multiplicative group modulo p, and x is an integer between 1 and $p - 1$. Computing f is easy using a divide-and-conquer approach (repeated squaring); however, inverting f is hard as it is essentially the *discrete logarithm problem* for which no efficient algorithm is known. One interesting property of this function is that it becomes a permutation if we fix p and g.

- *RSA:*
$$f(p, q, e, x) = (pq, e, x^e \bmod pq),$$
where p, q are primes, and e, x are integers satisfying $\gcd(x, \phi(pq)) = 1$ [1]. This function is also a permutation after fixing p, q and e. Inverting f becomes easy once d, the inverse of e modulo $\phi(pq)$, is given. One-way functions which are easy to invert given some special information are called *trapdoor functions*. In an RSA encryption system, e and d are used as the public key and private key respectively.

It is conjectured that for all of the above functions f,

$$HI_f(n) \geq 2^{n^\epsilon}$$

for some constant $\epsilon > 0$ which depends on f. Note that this statement is stronger than that required by a one-way function.

3.3 Existence of One-Way Functions

It is still an open question whether NP $\not\subseteq$ P/poly implies the existence of one-way functions. As mentioned previously, there are two ingredients we need to go from NP $\not\subseteq$ P/poly to the existence of one-way functions: average-case hardness of functions and efficient generation of "solved" hard instances. The second ingredient is in fact easy to obtain. For example, let f be a function defined by

$$f(\phi, x) = \begin{cases} \phi & \text{if } x \text{ satisfies } \phi \\ \rho & \text{otherwise} \end{cases}$$

where ϕ is a Boolean formula, x is an assignment for ϕ, and ρ is some trivial Boolean formula. Then, inverting f is equivalent to finding a satisfying assignment for ϕ, which is hard as long as P $\neq$ NP. Thus the key to showing the implication is to boost hardness from worst-case to high average-case. We encountered this before when we discussed how to go from worst-case hardness to high average-case hardness for functions in E by applying error-correcting codes with list decoding. However, the techniques for E fail in our case since they require exponential time to compute. Plausible directions to tackling this problem have been proposed. Some of them are discussed below.

3.3.1 Random Self-Reducibility

We use the discrete logarithm problem as an example. Recall the discrete logarithm problem:

> Given a prime p, a generator g of the multiplicative group modulo p, and an integer y $(1 \leq y < p)$, find x such that $y = g^x \bmod p$.

Consider the following algorithm, which uses an oracle that solves the discrete logarithm problem on some fraction of inputs.

1. Pick r uniformly at random.
2. Query the oracle to find s such that $g^r y = g^s \bmod p$.

[1] $\phi(n)$ is the *Euler totient function*. For different primes p and q, $\phi(pq) = (p-1)(q-1)$.

3. Output $s - r$.

Note that the distribution of $g^r y$ in step 2 is uniform provided that r is picked uniformly at random. If the oracle correctly solves a nontrivial fraction of inputs (say, $1/\operatorname{poly}(n)$) for all input size n, the above algorithm is a randomized algorithm for finding discrete logarithms. This shows that worst-case hardness of the discrete logarithm problem implies average-case hardness. Unfortunately, the discrete logarithm problem, as well as many other problems that exhibit random self-reducibility, are not complete for NP.

3.3.2 Concatenation

Given a function f, define

$$g(x_1, \ldots, x_k) = f(x_1) \circ \cdots \circ f(x_k),$$

where $\circ$ represents string concatenation. Inverting g on $(x_1, \ldots, x_k)$ requires inverting f on all of $x_1, \ldots, x_k$. If we can invert f only on a small fraction of inputs, we expect ourselves to be able to invert g on an even smaller fraction of inputs. Formally, suppose that we can only invert f on an ϵ fraction of inputs (of length n). Then, the fraction of inputs on which we can invert g is no more than ϵ^k. However, the best bound we have for worst-case hardness is $\epsilon > 1 - 2^{-n}$. Boosting hardness through concatenation thus requires k to be exponential in n, which is not feasible.

4 Next Lecture

Next time we will show that the existence of one-way functions allows us to construct cryptographic PRGs. We will also focus on authentication protocols based on zero-knowledge proof systems.