Computational Complexity November 26, 2018

Lecture 14 (Notes)

Lecturer: Ola Svensson *Scribes: Ola Svensson*

1 Introduction

Recall last lecture:

- Connection between PCP-Theorem and (hardness of) approximation algorithms.
- Clique is hard to approximate.
- Saving random bits using expanders.

Today: gentle introduction to cryptograph

- Crypto much older than computational complexity: ever since people began to write, the invented methods for secret writing. But the numerous methods of *encryption* or "secret writing" devised over the years had one common characteristic — sooner or later they were broken.
- Things changed in 1970s, when *modern cryptography* was born, whereby computational complexity was used to argue about the security of encryption schemes. Common assumptions are e.g. that factoring is a hard problem. It remains an important research problem to rely on weaker assumptions such as $\mathbf{P} \neq \mathbf{NP}$ to devise cryptography schemes. A difficulty is that crypto relies on a problem to be hard on "average" and not only in the worst case.
- Cryptography is a huge topic, and so naturally we will only cover a tiny bit of it. See e.g. textbook for further references.
- Today we do: perfect secrecy and its limitations, define one-way functions and pseudo-random generators, and see some of their applications.

2 Perfect secrecy and its limitations

The most basic setting studied is as follows:

- Alice wants to send a secret message x (known as *plaintext*) to Bob.
- The problem is that her adversary Eve is eavesdropping on the communication channel between Alice and Bob.
- Thus Alice will "scramble" the plaintext x using an encryption algorithm E to obtain a *ciphertext* $y = \mathsf{E}(x)$. And she sends y to Bob instead of x.
- Bob will then decode x from the ciphertext y using a *decryption algorithm* D.

The goal is of course that the encryption is good enough so as to make it impossible for Eve to decode the message. However, Bob is seeing the same information that Eve is, so in order to do something that Eve cannot, Bob has to know something that Eve does not. In the setting of *private key encryption* we assume that Alice and Bob share some secret string k (known as the key) that is chosen at random and shared secretly before hand.

The encryption scheme is thus defined by a pair of algorithms (E, D), each taking a key and a message (where we write the key input as a subscribpt), such that for every key k and plaintext x

$$\mathsf{D}_k(\mathsf{E}_k(x)) = x\,. \tag{1}$$

The above says that the scheme is correct in the sense that Bob can decode the message given the ciphertext. However, it does not say anything about the security. How can we define security? A first attempt may be to say that Eve should not be able to find x from y. However, this is often insufficient as this doesn't rule out the possiblity that Eve can find partial information about x. For example, if x is a top secret document consisting of 10 pages we should not allow Eve to recover a single page from this document. Shannon gave the following definition of secure private key encryption that ensures that Eve does not learn anything about x:

Definition 1 (Perfect secrecy) *Let (E, D) be an encryption scheme for messages of length m and with a key of length n satisfying (1). We say that (E, D) is* perfectly secret *if for every pair of messages $x, x' \in \{0,1\}^m$, the distributions $\mathsf{E}_{U_n}(x)$ and $\mathsf{E}_{U_n}(x')$ are identical, where U_n denotes the uniform distribution over $\{0,1\}^n$.*

Notice the above definition captures perfect secrecy as the ciphertext that Eve sees always has the same distribution, regardless of the plaintext and so she gets no information on the plaintext. There is a very simple scheme called one-time pad scheme to achieve perfect secrecy:

- To encrypt a message $x \in \{0,1\}^m$, Alice and Bob decides on a random key $k \in \{0,1\}^m$.
- Alice then sends $y = \mathsf{E}_k(x) = x \oplus k$ to Bob.
- Bob decrypts by $\mathsf{D}(y) = y \oplus k = x \oplus k \oplus k = x$.

It is not hard to see that this protocol is perfectly secret (an exercise). However, there are two problems:

- As the name (*one-time* pad) suggests, a key k can only be used once. If k us used twice for messages x and x' then Eve can calculate $\mathsf{E}(x) \oplus \mathsf{E}(x') = x \oplus x'$ which reveals nontrivial information about x and x'.
- The key is huge (as large as the message we want to communicate).

These drawbacks make the *one-time* pad impractical to use in modern settings. Moreover, one can show (homework) that no perfectly secret encryption scheme can use a key size shorter than the message size! We thus have to relax the perfect secrecy condition somehow.

3 Computational security and one-way functions

The main idea for relaxing the perfect secrecy is to devise schemes that are only required to be secure against eavesdroppers that are *efficient*, i.e., run in polynomial time.

We first start with a "negative" result. We show that it, even with this restriction on the eavesdropper, achieving perfect secrecy is impossible with small key sizes if $\mathbf{P} = \mathbf{NP}$.

Lemma 2 *Suppose that* $\mathbf{P} = \mathbf{NP}$. *Let* (E, D) *be any polynomial-time computable encryption scheme satisfying* (1) *with key shorter than the message. Then, there is a polynomial time algorithm* A *such that for every input length* m*, there is a pair of messages* $x_0, x_1 \in \{0,1\}^m$ *satisfying:*

$$\Pr_{b\in\{0,1\},k\in\{0,1\}^n}[A(\mathsf{E}_k(x_b)) = b] \geq 3/4\,,$$

where $n < m$ *denotes the key length.*

Proof

- Let (E, D) be an encryption for message of length m and with key length $n < m$.
- Let $S \subseteq \{0,1\}^*$ denote the support of $\mathsf{E}_{U_n}(0^m)$. Note that $y \in S$ if and only if $y = \mathsf{E}_k(0^m)$ for some k. Hence, if $\mathbf{P} = \mathbf{NP}$, then membership in S can be verified efficiently (by "guessing" the right k).
- Algorithm A will be very simple: on input y, it outputs 0 if $y \in S$ and it outputs 1 otherwise.
- We set $x_0 = 0^m$ and we will find some x_1 satisfying the statement of the lemma.
- If we let D_x denote the distribution $\mathsf{E}_{U_n}(x)$, then

$$\begin{aligned}\Pr_{b\in\{0,1\},k\in\{0,1\}^n}[A(\mathsf{E}_k(x_b)) = b] &= \frac{1}{2}\Pr[A(D_{x_0}) = 0] + \frac{1}{2}\Pr[A(D_{x_1}) = 1] \\ &= \frac{1}{2} + \frac{1}{2}\Pr[A(D_{x_1}) = 1]\,.\end{aligned}$$

- It thus suffice in finding an x_1 such that $\Pr[A(D_{x_1}) = 1] \geq \frac{1}{2}$ or, equivalently, $\Pr[D_{x_1} \in S] \leq 1/2$.
- To see this, define $S(x,k)$ to be 1 if $\mathsf{E}_k(x) \in S$ and 0 otherwise. Then

$$\mathbb{E}_{x\in\{0,1\}^m}\,\mathbb{E}_{k\in\{0,1\}^n}[S(x,k)] = \mathbb{E}_{k\in\{0,1\}^n}\,\mathbb{E}_{x\in\{0,1\}^m}[S(x,k)] \leq 1/2\,,$$

  where the last inequality follows from the fact that, for any fixed k, E_k is one-to-one and hence at most $2^n \leq 2^m/2$ of the x's can be mapped to the set S of size 2^n.
- Hence there must exist an x_1 such that $\mathbb{E}_{k\in\{0,1\}^n}[S(x_1,k)] \leq 1/2$ which is equivalent to $\Pr[D_{x_1} \in S] \leq 1/2$.

$\blacksquare$

Hence, it will be necessary to assume $\mathbf{P} \neq \mathbf{NP}$. We will relay on stronger assumptions (that one-way functions exist) and, as aforementioned, it is an important research problem to weaken the assumption. Before we define one-way functions we introduce a notation that simplifies notation:

Definition 3 (Negligible functions) *A function* $\epsilon : \mathbb{N} \to [0,1]$ *is called* negligible *if* $\epsilon(n) = n^{-\omega(1)}$, *i.e., for every constant* c *and sufficiently large* n, $\epsilon(n) < n^{-c}$*).*

Definition 4 (One-way functions) *A polynomial-time computable function* $f : \{0,1\}^* \to \{0,1\}^*$ *is a* one-way function *if for every probabilistic polynomial-time algorithm* A*, there is a negligible function* $\epsilon : \mathbb{N} \to [0,1]$ *such that for every* n*:*

$$\Pr_{x\in\{0,1\}^n}[A(f(x)) = x' \text{ s.t. } f(x') = f(x)] < \epsilon(n)\,.$$

In the exercise session, you will show that the existence of one-way functions implies $\mathbf{P} \neq \mathbf{NP}$. There are several candidate one-way functions that no one has yet been able to invert (see book for examples), and it is conjectured that they exist.

One reason that we are interested in one-way functions is because they can be used to design secure encryption schemes with keys much shorter than the message length.

Theorem 5 (Encryption from one-way function) *Suppose that one-way functions exist. Then for every $c \in \mathbb{N}$, there exists a computationally secure encryption scheme (E, D) using n-length keys for n^c-length messages.*

With computational secure, we informally mean that the ciphertext should not reveal any partial information about the plaintext to a polynomial-time eavesdropper. The formal definition turns out to be a little cumbersome and we omit it here (see the book, Exercise 9.9). A key component of the proof of Theorem 5 is the concept of pseudorandom generators that we introduce next.

4 Pseudorandom generators (PRGs)

The perfectly secure one-time pad encryption scheme XORed the message with a key with equal length. The problem with that protocol is the length of the key. The idea of the proof of Theorem 5 is to take a short completely random key k and stretch it to obtain a bit string of length m while still make it "look" random. But what does "looks random" mean? We might require that the output distribution of the "stretching" has some of the properties a truly random sequence have. For instance,

- Number of 0's and 1's differ by $O(\sqrt{m})$.
- The XOR of a fixed sized subset of the bits is unbiased.
- There is a probability $\Theta(1/m)$ that the output is a prime number.
- ...

We will in fact require that the output distribution has to "look" like the uniformly random distribution to *every* polynomial-time algorithm. Such a distribution is called *pseudorandom*. We now formally define pseudorandom generators:

Definition 6 *Let $G : \{0,1\}^* \to \{0,1\}^*$ be a polynomial-time function. We say that G is a* secure pseudorandom generator of stretch *$\ell(n) > n$ if $|G(x)| = \ell(|x|)$ for every $x \in \{0,1\}^*$ and for every probabilistic polynomial-time A, there exists a negligible function ϵ such that*

$$\left|\Pr[A(G(U_n)) = 1] - \Pr[A(U_{\ell(n)}) = 1]\right| \leq \epsilon(n)$$

for every $n \in \mathbb{N}$.

It is easy to see that the existence of pseudorandom generators imply the existence of one-way functions (an exercise). The other direction require much more work:

Theorem 7 (HILL'99) *If one-way functions exist, then for every $c \in \mathbb{N}$, there exists a secure pseudorandom generator with stretch $\ell(n) = n^c$.*

The next lecture will be devoted to the proof of a slight simplification of the above theorem. Here, we now describe how it implies Theorem 5 and then we also explain some other cool applications of pseudorandom generators.

The above definition states that it's infeasible for polynomial-time adversaries to distinguish between a completely random string of length $\ell(n)$ and a string that was generated by applying G to a much shorter random string of length n. Thus it's not hard to verify that Theorem 7 implies Theorem 5. An informal argument is as follows:

- If we modify the one-time pad encryption to generate its n^c-length random key by applying a secure pseudorandom generator with stretch n^c to a shorter key of length n, then a *polynomial-time* eavesdropper would not be able to tell the difference (without contradicting the security of the generator as per Definition 6).
- Therefore, since XORing the message with a random bit string is perfectly secure (one-time pad protocol) a polynomial-timeo eavesdropper cannot get any information about the message.

4.1 Other cool applications of PRGs

Apart from cryptography, pseudorandom generators have several interesting applications. We mention some here.

4.1.1 Derandomization

The existence of pseudorandom generators implies subexponential deterministic algorithms for **BPP**. Consider a language $L \in \mathbf{BPP}$ and let A be a randomized algorithm that decides L. We now reduce the number of random bits that A requires to obtain a deterministic algorithm D. Specifically, the algorithm D works by simply reducing the randomness requirement of A to n^ϵ by using a pseudorandom generator, and then enumerating over all the possible inputs for the pseudorandom generator.

4.1.2 Tossing coins over the phone and bit commitment

Suppose you (A) and your friend (B) want to toss an unbiased coin over the phone. How can you do it even if you do not completely trust your friend?

If only one of you tosses the coin there is nothing to prevent that party from lying. A first idea to fix this is as follows:

- A tosses a coin c_a and B tosses a coin c_b.
- The final coin is then the XOR: $c_a \oplus c_b$.

Notice that this kind of fixes the problem: Even if B doesn't trust A he knows that the output is completely random if his coin is random. However, there is another problem: the player that reveals his coinflip first is at a disadvantage because then the other player can just "adjust" his answer to get the desired final coin toss.

A solution is to use one-way permutations. A function $f : \{0,1\}^n \to \{0,1\}^n$ is a one-way permutation if it is a one-way function that is a bijection.

- First A chooses two strings x_A and r_A of length n and sends a message $(f(x_A), r_A)$.
- Then B selects a random bit b and sends it to A.
- Finally, A reveals x_A and they agree to use the XOR of b and $(x_A \bigodot r_A)$ as their coin toss.

Note that B can verify that X_A is the same as in the first message by applying f; therefore, A cannot change her mind after learning B's bit. (A's first message is a cryptographic commitment to the bit $(x_A \bigodot r_A)$. On the other hand, B cannot predict $x_A \bigodot r_A$ from A's first message (assuming the f is one-way as we will prove next lecture) and so she cannot bias her bit according to the choice of $x_A \bigodot r_A$.

4.1.3 Lower bounds for machine learning

In machine learning the goal is to learn a succinct function $f : \{0,1\}^n \to \{0,1\}$ from a sequence $(x_1, f(x_1)), (x_2, f(x_2)), \ldots,$ where the x_i's are randomly chosen examples from a distribution. This problem is hard if we assume a pseudorandom generator G: suppose f is the function that takes value 1 on strings generated by G and 0 on other strings. Then f should be hard to learn since we cannot distinguish G's output from truly random in polynomial-time.

This is the idea on a high level but we are still not completely satisfied with the result: clearly a random function f is also hard to learn. So in order to get a convincing hardness result for learning we would like f to be very easy to evaluate/describe but hard to learn. This is possible to achieve by using trapdoor pseudorandom generators and a function f that can be evaluated in parallel time cannot be learned in polynomial time. See e.g. the paper "Cryptographic Limitations on Learning Boolean Formulae and Finite Automata" by Kearns and Valiant.