18.425/6.875 Introduction to Cryptography

Lecture #2, September 16, 1991

1 Introduction

As of today, we do not even know how to prove a linear lower bound on the time required to solve an NP-complete problem. Thus, in our development of a theory of cryptography in the presence of a computationally bounded adversary we must resort to making assumptions about the existence of hard problems. In fact, an important current research topic in cryptography (on which much progress has been made in recent years) is to find the minimal assumptions required to prove the existence of “secure” cryptosystems.

In this lecture, we will discuss exactly what assumptions need to be made. Our assumptions should enable us to quickly generate instances of problems which are hard to solve for anyone other than the person who generated the instance. For example, it should be easy for the sender of a message to generate a ciphertext which is hard to decrypt for any adversary (naturally, in this example, it should also be easy for the intended recipient of the message to decrypt the ciphertext). To formally describe our assumptions (the existence of one way functions and trapdoor function) we first need to recall some complexity theory definitions.

2 Complexity Definitions

Complexity Class P

A language \( L \) is in P if and only if there exists a Turing machine \( M(x) \) and a polynomial function \( Q(y) \) such that on input string \( x \)

1. \( x \in L \) iff \( M \) accepts \( x \) (denoted by \( M(x) \)).

2. \( M \) terminates after at most \( Q(|x|) \) steps.

The class of languages P is classically considered to be those languages which are ‘easily computable’. We will use this term to refer to these languages and the term ‘efficient algorithm’ to refer to a polynomial time Turing machine.

---

1These notes were scribed by Atul Shrivastava.
### Complexity Class NP

A language \( L \) is in NP if and only if there exists a Turing machine \( M(x, y) \) and polynomials \( p \) and \( l \) such that on input string \( x \):

1. \( x \in L \Rightarrow \exists y \text{ with } |y| \leq l(|x|) \text{ such that } M(x, y) \text{ accepts and } M \text{ terminates after at most } p(|x|) \text{ steps.} \)
2. \( x \notin L \Rightarrow \forall y \text{ with } |y| \leq l(|x|), M(x, y) \text{ rejects.} \)

Note that this is equivalent to the (perhaps more familiar) definition of \( L \in \text{NP} \) if there exists a non-deterministic polynomial time Turing machine \( M \) which accepts \( x \) if and only if \( x \in L \). The string \( y \) above corresponds to the guess of the non-deterministic Turing machine.

### Complexity Class BPP

A language \( L \) is in BPP if and only if there exists a Turing machine \( M(x, y) \) and polynomials \( p \) and \( l \) such that on input string \( x \):

1. \( x \in L \Rightarrow \Pr_{|y| \leq l(|x|)}[M(x, y)\text{ accepts}] \geq \frac{2}{3}. \)
2. \( x \notin L \Rightarrow \Pr_{|y| \leq l(|x|)}[M(x, y)\text{ accepts}] \leq \frac{1}{3}. \)
3. \( M(x, y) \) always terminates after at most \( p(|x|) \) steps.

**Exercise 1** Show that if the constants \( \frac{2}{3} \) and \( \frac{1}{3} \) are replaced by \( \frac{1}{2} + \frac{1}{p(|x|)} \) and \( \frac{1}{2} - \frac{1}{p(|x|)} \) where \( p \) is any fixed polynomial then the class BPP remains the same.

**Hint:** Simply run the machine \( M(x, y) \) on “many” \( y \)'s and accept if and only if the majority of the runs accept. The magnitude of “many” depends on the polynomial \( p \).

We know that \( P \subseteq \text{NP} \) and \( P \subseteq \text{BPP} \). We do not know if these containments are strict although it is often conjectured to be the case. An example of a language known to be in BPP but not known to be in P is the language of all prime integers (that is, primality testing). It is not known whether BPP is a subset of NP.

### 2.1 Probabilistic Turing Machines

The class BPP could be alternatively defined using probabilistic Turing machines (probabilistic algorithms). A probabilistic polynomial time Turing machine \( M \) is a Turing machine which can flip coins as an additional primitive step, and on input string \( x \) runs for at most a polynomial in \( |x| \) steps. We could have defined BPP by saying that a language \( L \) is in BPP if there exists a probabilistic polynomial time Turing machine \( M(x) \) such that when \( x \in L \), the probability (over the coin tosses of the machine) that \( M(x) \) accepts
is greater than $\frac{2}{3}$ and when $x \notin L$ the probability (over the coin tosses of the machine) that $M(x)$ rejects is greater than $\frac{2}{3}$. The string $y$ in the previous definition corresponds to the sequence of coin flips made by the machine $M$ on input $x$.

From now on we will consider probabilistic polynomial time Turing machines as “efficient algorithms” (extending the term previously used for deterministic polynomial time Turing machines). We also call the class of languages in BPP “easily computable”. Note the difference between a non-deterministic Turing machine and a probabilistic Turing machine. A non-deterministic machine is not something we could implement in practice (as there may be only one good guess $y$ which will make us accept). A probabilistic machine is something we could implement in practice by flipping coins to yield the string $y$ (assuming of course that there is a source of coin flips in nature). Some notation is useful when talking about probabilistic Turing machines.

2.1.1 Notation For Probabilistic Turing Machines
Let $M$ denote a probabilistic Turing machine (PTM). $M(x)$ will denote a probability space of the outcome of $M$ during its run on $x$. The statement $z \in M(x)$ indicates that $z$ was output by $M$ when running on input $x$. $\Pr[M(x) = z]$ is the probability of $z$ being the output of $M$ on input $x$ (where the probability is taken over the possible internal coin tosses made by $M$ during its execution). $M(x, y)$ will denote the outcome of $M$ on input $x$ when internal coin tosses are $y$.

2.1.2 Different Types of Probabilistic Algorithms
Monte Carlo algorithms and Las Vegas algorithms are two different types of probabilistic algorithms. The difference between these two types is that a Monte Carlo algorithm always terminates within a polynomial number of steps but its output is only correct with high probability whereas a Las Vegas algorithm terminates within an expected polynomial number of steps and its output is always correct. Formally, we define these algorithms as follows.

**Definition 1** A Monte Carlo algorithm is a probabilistic algorithm $M$ for which there exists a polynomial $P$ such that for all $x$, $M$ terminates within $P(|x|)$ steps on input $x$. Further,

$$\Pr[M(x) \text{ is correct }] > \frac{2}{3}$$

(where the probability is taken over the coin tosses of $M$).

A Las Vegas algorithm is a probabilistic algorithm $M$ for which there exists a polynomial $p$ such that for all $x$, $E(\text{running time}) = \sum_{t=1}^{\infty} t \cdot \Pr[M(x) \text{ takes exactly } t \text{ steps}] < p(|x|)$. Further, the output of $M(x)$ is always correct.
All Las Vegas algorithms can be converted to Monte Carlo algorithms but it is unknown whether all Monte Carlo algorithms can be converted to Las Vegas algorithms. Some examples of Monte Carlo algorithms are primality tests such as Solovay-Strassen (see [4]) or Miller-Rabin (see [3]) and testing the equivalence of multivariate polynomials and some examples of Las Vegas algorithms are computing square roots modulo a prime \( p \), computing square roots modulo a composite \( n \) (if the factors of \( n \) are known) and primality tests based on elliptic curves (see [1] or [2]).

2.2 Non-Uniform Polynomial Time \( P / \text{poly} \)

An important concept is that of polynomial time algorithms which can behave differently for inputs of different size, and may even be polynomial in the size of the input (rather than constant as in the traditional definition of a Turing machine).

**Definition 2** A non-uniform algorithm \( A \) is an infinite sequence of Turing machines \( \{M_i\} \) (one for each input size \( i \)) such that on input \( x \), \( M_{|x|}(x) \) is run. We say that \( A(x) \) accepts if and only if \( M_{|x|}(x) \) accepts. We say that \( A \) is a polynomial time non-uniform algorithm if there exist polynomials \( P \) and \( Q \) such that \( M_{|x|}(x) \) terminates within \( P(|x|) \) steps and the size of the description of \( M_i \) (according to some standard encoding of all Turing machines) is bounded by \( Q(|i|) \).

**Definition 3** We say that a language \( L \) is in \( P / \text{poly} \) if \( \exists \) a polynomial time non-uniform algorithm \( A = \{M_i\} \) such that \( x \in L \iff M_{|x|}(x) \) accepts.

An alternative way to define \( P / \text{poly} \) is to use the notion of a Turing machine with advice. We consider this next.

2.2.1 Turing Machines With Advice

**Definition 4** A non-uniform polynomial time machine with advice is a pair \((M, a)\) where \( M(x, y) \) is a polynomial time Turing machine and \( a = a_1, \ldots, a_n, \ldots \) is an infinite sequence for which there exists a polynomial \( p \) such that \( |a_n| < p(n) \) for every \( n \). On input string \( x \), the machine \( M \) is run on inputs \( x \) and \( a_{|x|} \). We denote the result of the computation by \( M(x, a_{|x|}) \). We say that \((M, a)\) accepts language \( L \) if on input string \( x \)

1. \( x \in L \Rightarrow M(x, a_{|x|}) \) accepts.
2. \( x \not\in L \Rightarrow M(x, a_{|x|}) \) does not accept.

We let \( P / \text{poly} \) be the set of all languages \( L \) for which there exists a non-uniform polynomial time machine with advice \((M, a)\) such that \((M, a)\) accepts \( L \).

There are several relationships known about \( P / \text{poly} \). It is clear that \( P \subset P / \text{poly} \) and it has been shown by Adleman that \( \text{BPP} \subset P / \text{poly} \). (The proof uses the idea that if
If $L \in \text{BPP}$ then there must be for every size $n$ of inputs, one fixed sequence of coin tosses $a_n$ such that the probabilistic Turing machine $M(x, a_n)$ accepts $x$ where $|x| = n$ if and only if $x \in L$. Thus, for every input length $n$ one can let string $a_n$ be the advice given to Turing machine $M$ which on input $x$ simply simulates $M(x, a_n)$. Moreover, $\exists L \in \text{P/poly}$ such that $L \not\in \text{NP}$. Hence, $\text{P/poly} \not\subset \text{NP}$. However, the relationship in the opposite direction is unknown; that is, it is not known if there exists a language $L \in \text{NP}$ such that $L \not\in \text{P/poly}$.

We will use the term ‘efficient non-uniform algorithm’ to refer to a non-uniform polynomial time algorithm and the term ‘efficiently non-uniform computable’ to refer to languages in the class $\text{P/poly}$.

3 Adversaries

We will have two kinds of adversaries. The first is the (uniform) adversary (we will usually drop the “uniform” when referring to it). A **uniform adversary** is any polynomial time probabilistic algorithm. A **non-uniform adversary** is any non-uniform polynomial time algorithm. Thus, the adversary can use different algorithms for different sized inputs. Clearly, the non-uniform adversary is stronger than the uniform one. Thus to prove that “something” is “secure” even in presence of a non-uniform adversary is a better result than only proving it is secure in presence of a uniform adversary.

Assumptions To Be Made

The weakest assumption that must be made for cryptography in the presence of a uniform adversary is that $\text{P} \neq \text{NP}$. Namely, $\exists L \in \text{NP}$ such that $L \not\in \text{P}$. Unfortunately, this is not enough as we assumed that our adversaries can use probabilistic polynomial time algorithms. So we further assume that $\text{BPP} \neq \text{NP}$. Is that sufficient? Well, we actually need that it would be hard for an adversary to crack our systems most of the time. It is not sufficient that our adversary can not crack the system once in a while. Assuming that $\text{BPP} \neq \text{NP}$ only means that there exists a language in $L \in \text{NP}$ such that every uniform adversary makes (with high probability) the wrong decision about infinitely many inputs $x$ when deciding whether $x \in L$. These wrong decisions, although infinite in number, may occur very infrequently (such as once for each input size).

We thus need yet a stronger assumption which will guarantee the following. There exists a language $L \in \text{NP}$ such that for every sufficiently large input size $n$, every uniform adversary makes (with high probability) the wrong decision on infinitely many inputs $x$ of length $n$ when deciding whether $x$ is in $L$. Moreover, we want it to be possible, for every input size $n$, to generate input $x$ of length $n$ such that with high probability every uniform adversary will make the wrong decision on $x$.

The assumption that will guarantee the above is the existence of (uniform) one way
functions. The assumption that would guarantee the above in the presence of non-uniform adversary is the existence non-uniform one way functions. Let us proceed to define one way functions.

4 One Way Functions

We define here strong and weak one way functions. In what follows, when we refer to a function as being one way we implicitly mean that it is a strong one way function. However, this precaution is not necessary because, as we will soon show, the problem of finding a weak one way function is equivalent to the problem of finding a strong one way function.

Definition 5 A function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is called a strong one way function if

1. $f(x)$ is computable in polynomial time.
2. For all polynomials $Q$ and for all uniform adversaries $A \exists k_0$ such that $\forall k > k_0$
   \[ Pr[A(f(x)) \in f^{-1}(f(x))] < \frac{1}{Q(k)} \]
   (where the probability is taken over the coin tosses of $A$ and the choices of $x \in \{0, 1\}^k$).

Remark Because $f(x)$ is computable in polynomial time, $\exists c$ such that $\forall x$, $|f(x)| < |x|^c$.
Furthermore, we shall restrict our attention to interesting one way functions; that is, $\exists c$ such that $|f(x)| \geq |x|^c$. Otherwise, it would be very easy to invert $f$.

Remark A non-uniform (strong) one way function is defined exactly as above except that the adversary $A$ in condition 2 is a non-uniform adversary.

Now, Consider for example the function $f : \mathbb{Z} \times \mathbb{Z} \mapsto \mathbb{Z}$ where $f(x, y) = x \cdot y$. This function can be easily inverted on at least half of its outputs (namely, on the even integers) and thus is not a strong one way function. Still, we said in the first lecture that $f$ is hard to invert when $x$ and $y$ are primes of roughly the same length which is the case for a polynomial fraction of the $k$-bit composite integers. This motivates the following definition of a weak one way function.

Definition 6 A function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ is called a weak one way function if

1. $f(x)$ is computable in polynomial time.
2. There is a polynomial $Q$ such that for all uniform adversaries $A$, $\exists k_0$ such that

$$\forall k > k_0 \quad \Pr[A(f(x)) \not\in f^{-1}(f(x))] \geq 1 - \frac{1}{Q(k)}$$

(where the probability is taken over the coin tosses of $A$ and the choices of $x \in \{0,1\}^k$).

Consider once again the function $f(x, y) = x \cdot y$ where $x, y \in \mathbb{Z}$. Since the probability that an $k$-bit integer $x$ is prime is approximately $1/k$, we get the probability that both $x$ and $y$ such that $|x| = |y| = k$ are prime is approximately $1/k^2$. Thus, for all $k$, about $1 - \frac{1}{k^2}$ of the inputs to $f$ of length $2k$ are prime pairs of equal length. It is believed that no adversary can invert $f$ when $x$ and $y$ are primes of the same length with good (bigger than $\frac{1}{p(|x|)}$ for any polynomial $p$) success probability. Under this belief, $f$ is a weak one way function (as condition 2 in the above definition is satisfied for $Q(k) = O(k^2)$).

Clearly, it would be more desirable to have strong one way functions. But, what if only weak one way functions exist? Theorem 1 tells us that the existence of a weak one way function implies the existence of a strong one way function.

**Theorem 1**  Weak one way functions exist if and only if strong one way functions exist.

**Sketch of Proof:**
Let’s first outline the proof to get some intuition for it. A detailed and rigorous proof will be presented in the next lecture.

$(\Rightarrow)$ By definition, a strong one way function is a weak one way function.

$(\Leftarrow)$ Assume that $f$ is a weak one way function such that $Q$ is the polynomial in condition 2 in the definition of a weak one way function. Define the function

$$f_1(x_1 \ldots x_N) = f(x_1) \ldots f(x_N)$$

where $N = 2kQ(k)$ and each $x_i$ is of length $k$.

We claim that $f_1$ is a strong one way function. Since $f_1$ is a concatenation of $N$ copies of the function $f$, to correctly invert $f_1$, we need to invert $f(x_i)$ correctly for each $i$. We know that every adversary has a probability of at least $\frac{1}{Q(k)}$ to fail to invert $f(x)$ (where the probability is taken over $x \in \{0,1\}^k$ and the coin tosses of the adversary), and so intuitively, to invert $f_1$ we need to invert $O(kQ(k))$ instances of $f$. The probability that the adversary will fail for at least one of these instances is extremely high.
The formal proof will take the form of a reduction; that is, we will assume for contradiction that \( f_1 \) is not a strong one way function and that there exists some adversary \( A_1 \) that violates condition 2 in the definition of a strong one way function. We will then show that \( A_1 \) can be used as a subroutine by a new adversary \( A \) that will be able to invert the original function \( f \) with probability better than \( 1 - \frac{1}{Q(|x|)} \) (where the probability is taken over the inputs \( x \in \{0,1\}^k \) and the coin tosses of \( A \)). But this will mean that \( f \) is not a weak one way function and we have derived a contradiction.

References


