CS 601.642/442
Modern Cryptography
Lecture Notes

Abhishek Jain

September 13, 2020

THE DOCUMENT IS UNDER CONTINUAL UPDATE

Contents

List of Figures

Acknowledgment

Thanks to the students of CS 600.442 (Fall 2016) (Alex Badiceanu, Ke Wu, David Li, Yeon Woo Kim, Aarushi Goel, Jesse Fowers, Neil Fendley, Katie Chang, Alishah Chator, Arka Rai Choudhuri, Cheng-Hao Cho) for scribing the original lecture notes.

Chapter 1

Introduction

These are an edited collection of the lecture notes for the *Modern Cryptography* course. The notes are meant to complement the lecture, and not meant as a replacement to the actual lectures.

While we shall attempt to be as formal as possible in these notes, formalism is not the main goal of the course. Often, where obvious, we shall forgo some of the formalism for what we believe to be a simpler exposition. But we shall make it a point to provide sufficient references for the complete formal exposition.

Structure. The structure of the notes will follow the structure of the course. We shall start with a recap of some of the important mathematical concepts that will be essential to the course. For the main content, we start with one of the most important conceptual building blocks of cryptography, one-way functions.

We shall then discuss the importance of randomness in cryptography, and describe what it means to be "almost random". This will motivate how we can take a small amount of randomness and generate large amounts of "almost randomness".

We then move to encryption, and define what it means for encryption to be secure. We shall see how the topics we've covered up to this point will help us construct secure encryption.

Following this we shall see how cryptography allows us to achieve a digital analogue of signatures that will be broadly referred to as authentication. We provide both definitions and constructions based on the tools developed thus far.

We shall then move to more advanced topics covering proof systems that are leak no information (zero-knowledge), secure-computation and encryption schemes that are resistant to tampering of ciphertexts.

Errata. There might be errors of various forms that may have crept in during the editing of these notes, and we would be grateful if you could send an email to Arka Rai Choudhuri achoud@cs.jhu.edu pointing them out.

Chapter 2

Preliminaries

In this chapter, we're going to discuss some of the essential mathematics that we will require through the course. This will also provide an opportunity for us to establish notation moving forward. It should be noted that we're only going to cover these concepts at a very high level, and we've provided references at the end of this chapter if you feel the need to have a more in-depth coverage of these topics.

A large section of this chapter was motivated by Boaz Barak's wonderful notes for his cryptography course [Bar].

It is not expected for a student to read this chapter in its entirety before proceeding with the rest of the notes. In fact, the student is encouraged to glance through this chapter to understand notation, and come back when certain concepts from other chapters are unclear.

2.1 Sets

We start with one of the simplest notions in mathematics, *sets*, which is a collection of *distinct* objects. Our first example is the infinite set of natural numbers,

$$\mathbb{N} \coloneqq \{1, 2, 3, \cdots\} .$$

This also gives us the opportunity to establish our first notation of ":=", which we will use as the assignment operator. This is to be contrasted with "=", which is the operator used to establish equality.

The most common set we shall encounter in the course is the finite set $\{0, 1\}$. This will often be referred to as an *alphabet* as we shall use it to build strings

$$\{0,1\}^2 \coloneqq \{00, 01, 10, 11\}$$
$$\{0,1\}^* \coloneqq \{\epsilon, 0, 1, 00, 01, 10, 11, 000 \cdots\}$$

Size. We shall restrict our discussion of size to *finite sets*. We denote by $|S|$ the size of the set S, defined to be the number of elements in the set.

Example 1 *Let the set S be defined as follows:*

$$S \coloneqq \{1, 01, 10, 100, 010, 001\}.$$

Then,

$$|S| = 6$$

Note that by definition, a set only consists of *distinct* elements, and thereby it suffices to define the size simply as the number of elements.

Membership. For an element x, we shall indicate by $x \in S$ if x is in the set S. If not, we shall indicate it by $x \notin S$.

2.1.1 Operations

Below we describe the common set operations:

Union. The union of two sets A and B, denoted by $A \cup B$ is defined as

$$A \cup B \coloneqq \{x \mid x \in A \text{ OR } x \in B\},$$

to be the set of elements that are present in either A or B.

The above shorthand notation is called the *set-builder notation*, that describes the properties an element x must satisfy to be a part of the set $A \cup B$. We shall use the *set-builder notation* frequently in this course.

Intersection The intersection of two sets A and B, denoted by $A \cap B$ is defined as

$$A \cap B \coloneqq \{x \mid x \in A \text{ AND } x \in B\},$$

to be the elements present only in both A and B.

Difference The set difference of A and B, denoted by $A \setminus B$, is defined as

$$A \setminus B \coloneqq \{x \mid x \in A \text{ AND } x \notin B\},$$

to be the elements present in A but not in B.

Note that the for set difference, the order of the two sets matter, and $A \setminus B \neq B \setminus A$. Think of a simple example to illustrate this.

Complement The complement of a set A is defined with respect to the *universe*, denoted by U. The universe denotes all possible elements which can be used to construct the set A. The complement of A, denoted by $\overline{A}$, is defined as

$$\overline{A} \coloneqq \{x \mid x \notin A\},$$

to be the elements not present in A.

These four common set operations are illustrated in Figures 2.1,2.2, 2.3 and 2.4.

For our context, it is easiest of think of the universe to be $\{0,1\}^*$, the set of all binary strings.

There are standard relations between these set operations, and we encourage the reader to work these out if they aren't familiar with them.

Cartesian product. The cartesian product of sets A and B, denoted by $A \times B$ is defined as

$$A \times B \coloneqq \{(a,b) \mid a \in A \text{ AND } b \in B\}$$

is the set of all *ordered pairs* (a,b) when $a \in A$ and $b \in B$. To differentiate unordered sets from an ordered tuple, we will represented ordered tuples by brackets $(\cdot)$.

Since we are talking about ordered tuples, $A \times B$ and $B \times A$ give rise to different sets since $(a,b) \neq (b,a)$ when a and b are distinct.

Subsets. For sets A and B, A is a subset of B, denoted by $A \subseteq B$, if every element of A is also an element of B.

Two sets A and B are equal if *both* $A \subseteq B$ and $B \subseteq A$. When arguing two sets are equal, this is the go-to method for a proof.

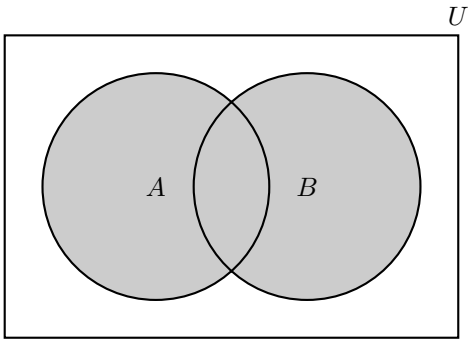


Figure 2.1: $A \cup B$

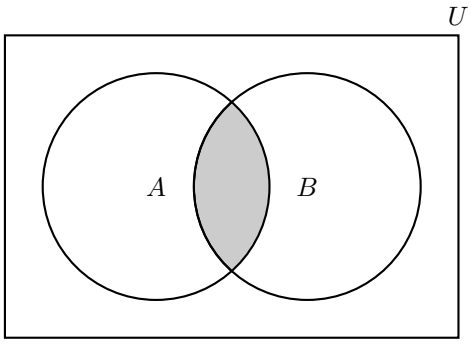


Figure 2.2: $A \cap B$

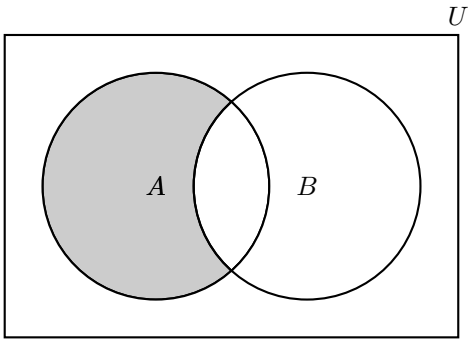


Figure 2.3: $A \setminus B$

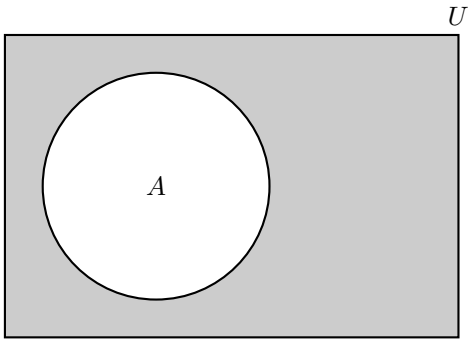


Figure 2.4: $\overline{A}$

2.2 Relations and Functions

2.2.1 Relation

A relation R from A to B is a subset of the Cartesian product $A \times B$. Consider the following relation R_1 from $A \coloneqq \{x_1, x_2, x_3, x_4\}$ to $B \coloneqq \{y_1, y_2, y_3, y_4\}$,

$$R_1 \coloneqq \{(x_1, y_2), (x_1, y_3), (x_2, y_4), (x_3, y_3), (x_4, y_1)\}$$

The *domain* of a relation is the subset of A that appear as the first component in a relation; and the *range* is the subset of B that appears as the second component in the relation. We shall typically consider relations where the domain is the entire set A. The set B is referred to as the co-domain of the relation.

2.2.2 Functions

A function $f : A \mapsto B$ is a relation from A to B with the additional restriction that each element in the domain A appears in exactly one ordered pair in the relation. The relation R_1 is not a function since x_1 appears in two ordered pairs. Consider R_2 over the same sets A and B.

$$R_s \coloneqq \{(x_1, y_2), (x_2, y_1), (x_3, y_3), (x_4, y_1)\}$$

The relations R_1 and R_2 are illustrated in Figures 2.5 and 2.6.

In function terminology, the elements from A are referred to as inputs, and the corresponding second element in the pair from B to the output. When we consider boolean functions, it is common to represent functions by a truth table, i.e. a row corresponding to each possible input and the corresponding output of the function.

Example 2 *What are the total number of boolean functions with n_1 inputs and n_2 outputs?*

Solution Let us consider first the simple case of a single output, and we shall see how to extend this to multiple outputs. As discussed above, every function can be represented by the corresponding truth table. For a function with n_1 inputs, any truth table will have 2^{n_1} rows.

For each row, since there is a single output, we can specify the output for that row to be either 0 or 1. Specifying such a value for each row determines the function. The total number of ways one can fill in this column then determines the number of functions. Hence the total number of functions with a single bit of output is $2^{2^{n_1}}$.

x_1	x_2	$\cdots$	x_{n_1}	y
0	0	$\cdots$	0	r_1
0	0	$\cdots$	1	r_2
		$\vdots$		
1	1	$\cdots$	0	r_{N-1}
1	1	$\cdots$	1	r_N

Now let us extend this to multiple output bits. The first important point to note is that the number of rows in the truth table remains unchanged.

x_1	x_2	$\cdots$	x_{n_1}	y_1	$\cdots$	y_{n_2}
0	0	$\cdots$	0	$r_{1,1}$	$\cdots$	$r_{n_2,1}$
0	0	$\cdots$	1	$r_{1,2}$	$\cdots$	$r_{n_2,2}$
			$\vdots$			
1	1	$\cdots$	0	$r_{1,N-1}$	$\cdots$	$r_{n_2,N-1}$
1	1	$\cdots$	1	$r_{1,N}$	$\cdots$	$r_{n_2,N}$

$$\underbrace{2^{2^{n_1}} \times \cdots \times 2^{2^{n_1}}}_{n_2 \text{times}} = 2^{n_2 2^{n_1}}$$

□

We now describe some special cases of functions below.

Injective Function. *Injective* functions or 1-1 functions are functions where each input has a distinct output, i.e. there does not exist $x, x' \in A$ such that $f(x) = f(x')$. This immediately requires $|B| \geq |A|$.

Surjective Function. A function is *surjective* if the co-domain B is the same as the range, i.e. every element of B is an output of the function for some element of A. This requires $|A| \geq |B|$.

Bijective Function. A function that is both *injective* and *surjective* is called a *bijective* function.

Permutation. A permutation on a set A is defined to be a *bijection* from A to itself. It is common to denote a permutation as $\Pi : A \mapsto A$.

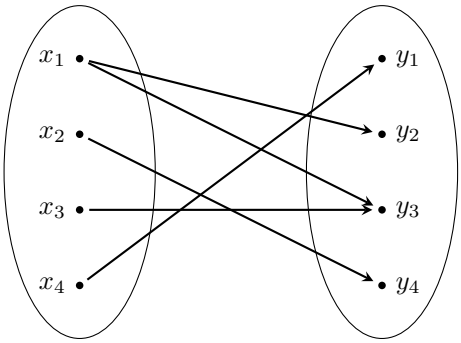


Figure 2.5: An example relation R_1

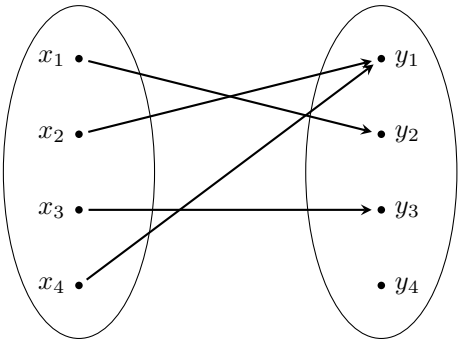


Figure 2.6: An example function R_2

2.2.3 Logical Operations

We describe below the four most common logical/boolean operations. While the AND, OR and XOR gate have been described with two inputs, they can easily be extended to multiple inputs.

AND. AND gate

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

OR. OR gate

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

XOR. XOR gate

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

NOT.

x	$\overline{x}$ (or $\neg x$)
0	1
1	0

2.2.4 Quantification.

We will often want to make a claim about a variable by either claiming a statement holds for all values that the variable takes, or there is at least one value satisfying the statement. The following are largely taken from the notes in [Lov], and you should take a look there for further details.

Universal Quantification.

$$\forall x \in A,\ P(x)$$

which indicates that for all x in the set A, the statement $P(x)$ is true.

Existential Quantification.

$$\exists x \in A,\ P(x)$$

which indicates that there is at least one x in A such that the statement $P(x)$ is true.

Nesting Quantifiers. We can nest the above quantifiers in an arbitrary manner. For instance, the following is a valid nesting

$$\forall x_1 \in A_1, \forall x_2 \in A_2, \exists x_3 \in A_3, \forall x_4 \in A_4\ P(x_1, x_2, x_3, x_4).$$

Given the fact that the nesting can be arbitrary, it is important to ask if there order of quantification matters. If the quantifiers are the same, then the order of nesting does not matter. Therefore, $\forall x_1 \in A_1, \forall x_2 \in A_2\ P(x_1, x_2)$ and $\forall x_2 \in A_2, \forall x_1 \in A_1\ P(x_1, x_2)$ are equivalent, and the same is true for the existential quantifiers. But things aren't as simple when the quantifiers are different.

Remark 1 (Order of Quantification) *When the nested quantifiers are different, the order of quantifiers matters. We illustrate this with an example below.*

Example 3 *Consider the two following statements*

1. $\forall x \in \mathbb{Z}, \exists y \in \mathbb{Z}$ *s.t.* $x + y = 4$
2. $\exists x \in \mathbb{Z},$ *s.t.* $\forall y \in \mathbb{Z}\ x + y = 4$

The first statement is true, since for every fixed x, we can set y to be $4 - x$. Thus existence of such a $y \in \mathbb{Z}$ is guaranteed. On the other hand, the second statement says that there must be a single x such that for every value of $y \in \mathbb{Z}$, $x + y = 4$. It is clear to see that this statement cannot be true.

In fact, the second ordering is a stronger claim than the first.

Negation of Quantifiers. When negating a quantified statement, negate all the quantifiers first, from left to right (keeping the same order), then negate the statement. By negating quantifiers we mean swapping $\forall$ and $\exists$. Below are few examples of negation

1. $\neg\left(\forall x \in A,\ P(x)\right) \iff \exists x \in A,\ \neg P(x)$
2. $\neg\left(\exists x \in A,\ P(x)\right) \iff \forall x \in A,\ \neg P(x)$
3. $\neg\left(\exists x \in A, \forall y \in B,\ P(x, y)\right) \iff \forall x \in A, \exists y \in B,\ \neg P(x, y)$
4. $\neg\left(\forall x \in A, \exists y \in B,\ P(x, y)\right) \iff \exists x \in A, \forall y \in B,\ \neg P(x, y)$

Here, $\iff$ is the notation for *if and only if* that connects two statements, where either both statements are true or both are false.

2.3 Reductio ad Absurdum

Reduction to absurdity, or proof by contradiction is a common proof technique that we shall employ throughout this course. We start with the statement we want to prove, and then assume that the statement is false, and then derive as a consequence a statement we believe to be false.

The following is an example taken from Boaz Barak's lecture notes:

Theorem 1 *There are infinitely many primes.*

Proof Let us assume for that the above statement is indeed false and there are a finite number of primes. Let us denote the primes by $p_1, \cdots, p_N$ with N indicating the total number of primes.

Consider the following number,

$$P = p_1 \cdot p_2 \cdots p_N + 1,$$

i.e. P is one greater than the product of all the primes. It is clear that none of the primes, $p_1, \cdots, p_N$ divide P, since the remainder with respect to all of them is 1.

Therefore, P has only two factors 1 and P, establishing it as a prime. But P is clearly not in the set of finite primes, therefore a contradiction to our assumption of a finite set of primes.

Since the same argument can be applied to any set of finite primes, it must be the case that the number of primes are infinite. □

Another common example of this type of proof technique is to prove that $\sqrt{2}$ cannot be expressed as an irreducible fraction, thereby establish that it is irrational.

2.4 Graphs

We describe below the notations we shall use to describe graphs, and their corresponding representation.

Definition 1 (Graphs) *A Graph $G = (V, E)$ where V is a set of vertices and E is a set of edges s.t. $|V|$ = n, $|E|$ = m.*

Example 4

$$V \coloneqq \{v_1, v_2, v_3, v_4, v_5, v_6\}$$
$$E \coloneqq \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_1\}, \{v_4, v_5\}, \{v_5, v_6\}\}$$

The order of the graph is the number of vertices, $|V| = n$, and the size of the graph is the number of edges, $|E| = m$.

To use graphs, we shall need to find a way to represent them. For this course, we shall find it convenient to represent graphs by an *adjacency matrix*.

Adjacency Matrix. Any graph can be represented as an adjacency matrix. The number of rows and columns of this matrix is the same as the number of vertices in the graph. A value of 1 at a given position represents the presence of an edge between the vertices corresponding to the row and column. More specifically:

Definition 2 (Adjacency Matrix) *A graph $G = (V, E)$ with $|V| = n$, can be represented as an $n \times n$ adjacency matrix M_G of boolean values such that:*

$$M_G[i, j] = \begin{cases} 1 & \text{if } (i, j) \text{ or } (j, i) \in E \\ 0 & \text{otherwise} \end{cases}$$

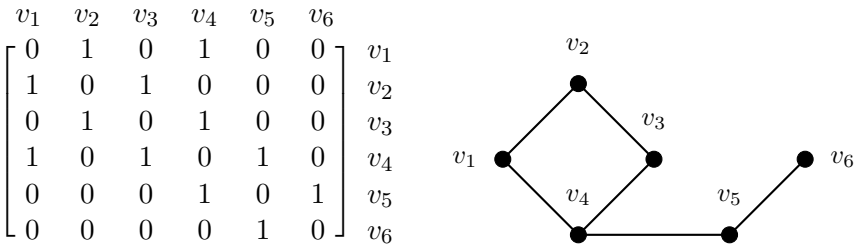


Figure 2.7: M_G and the corresponding graph G

2.5 Probability

Throughout this course, we shall see the importance of randomness, in that it permeates every aspect of cryptography. In this section, we shall review some of the relevant material for discrete probability.

Sample Space: Sample space S of a probabilistic experiment is the set of all possible outcomes of the experiment. A couple points of note:

- We will only consider finite sample spaces
- In most cases, the sample space will be the set $\{0, 1\}^k$ of size 2^k

Probability Distribution: With the sample space, we now define a distribution which assigns probabilities to this sample space.

Definition 3 (Distribution) *X is a **distribution** over a sample space $\mathcal{S}$ if it assigns a probability p_s to the element $s \in \mathcal{S}$ such that*

$$\sum_{s \in \mathcal{S}} p_s = 1.$$

A common distribution is a *uniform distribution* where each element in the sample space is assigned the same probability $\frac{1}{|S|}$. For example, when the sample space is the set $\{0,1\}^k$, this distribution is denoted by U_k with each k-bit string assigned probability $\frac{1}{2^k}$.

Sampling From Distribution: We say we sample an element x from the distribution X if each element in S is picked proportional to the probability defined by the distribution X. We denote this by $x \leftarrow\!\!\$\, X$. For uniform distribution, we shall find it convenient to denote sampling from the uniform distribution by $x \leftarrow\!\!\$\, \{0,1\}^k$ to indicate sampling uniformly from the set $\{0,1\}^k$.

Event: Event is a subset of the sample space. Let $E \subset S$ be an event, the probability of the event E, denoted by $\Pr[E]$, is defined as

$$\Pr[E] \coloneqq \sum_{s \in E} p_s$$

Union Bound: If S is a sample space and $A, A' \subseteq S$, then the probability that either A or A' occurs is $\Pr[A \cup A'] \leq \Pr[A] + \Pr[A']$

Random Variables: A random variable is a function that maps elements of the sample space to another set. It assigns values to each of the experiment's outcomes.

Example 5 *In the uniform distribution $\{0,1\}^3$, Let Random Variable N denote the number of $1s$ in the chosen string, i.e., for every $x \in \{0,1\}^3$, $N(x)$ is the number of $1s$ in x.*

$$\Pr_{x \leftarrow\!\!\$\, \{0,1\}^3}[N(x) = 2] = \frac{3}{8}$$

Expectation: The expectation of a random variable is its weighted average, where the average is weighted according to the probability measure on the sample space. The expectation of random variable N is defined as:

$$\mathbb{E}[N] = \sum_{x \in S} N(x) \cdot \Pr_{y \xleftarrow{\$} S}[y = x] \tag{2.1}$$

Here S is the sample space and $\Pr_{y \xleftarrow{\$} S}[y = x]$ is the probability of obtaining x when sampling from S.

Example 6 *If N is defined as in the above example,*

$$\mathbb{E}[N] = 0 \cdot \frac{1}{8} + 1 \cdot \frac{3}{8} + 2 \cdot \frac{3}{8} + 3 \cdot \frac{1}{8} = 1.5$$

Expectation is a linear function, i.e.,

$$\mathbb{E}[N + M] = \mathbb{E}[N] + \mathbb{E}[M]$$

Variance: Variance of a random variable N is defined as the expectation of the square of the distance of N from its expectation.

$$\mathsf{Var}[N] = \mathbb{E}\left[(N - \mathbb{E}[N])^2\right] \tag{2.2}$$

If N is defined as in the first example,

$$\begin{aligned}\mathsf{Var}[N] &= (0 - 1.5)^2 \cdot \frac{1}{8} + (1 - 1.5)^2 \cdot \frac{3}{8} + (2 - 1.5)^2 \cdot \frac{3}{8} + (3 - 1.5)^2 \cdot \frac{1}{8} \\ &= 0.75\end{aligned}$$

Variance is a measure of how spread out the values in a distribution are. A low variance means the outcomes will usually be very close to one another.

Standard Deviation: Standard deviation of N is the square root of $\mathsf{Var}[N]$

Conditional Probability: The conditional probability of event B in relation to event A is the probability of event B occurring when we know that A has already occurred.

$$\Pr[B \mid A] = \frac{\Pr[A \cap B]}{\Pr[A]} \tag{2.3}$$

Example 7 *Drawing Kings from a deck of cards. Event A is drawing a King first, and Event B is drawing a King second.*

$$\Pr[A] = \frac{4}{52}$$

$$\Pr[B|A] = \frac{3}{51}$$

Law of Total Probability: Throughout the course we will constantly use the law of total probability with regards to conditional probability.

Let $B_1, B_2, \cdots, B_n$ be a disjoint partition of a sample space S (i.e. $\forall i, j\ B_i \cap B_j = \emptyset$). Then for any event E,

$$\Pr[E] = \sum_{i=1}^{n} \Pr[E \cap B_i] = \sum_{i=1}^{n} \Pr[E \mid B_i\,] \Pr[B_i]$$

Let's look at a very simple example.

Example 8 *Consider any event E. Let us sample a single bit b uniformly at random. B_1 is the event that $b = 0$, and B_2 the event that $b = 1$. We can write, the probability of event E as*

$$\begin{aligned}\Pr[E] &= \Pr[E \mid b = 0\,] \Pr_{b \leftarrow\$ \{0,1\}}[b = 0] + \Pr[E \mid b = 1\,] \Pr_{b \leftarrow\$ \{0,1\}}[b = 1] \\ &= \Pr[E \mid b = 0\,] \cdot \frac{1}{2} + \Pr[E \mid b = 1\,] \cdot \frac{1}{2} \\ &= \frac{1}{2} \cdot (\Pr[E \mid b = 0\,] + \Pr[E \mid b = 1\,])\end{aligned}$$

Independent Events: We say that B is independent from A if $\Pr[B \mid A\,] = \Pr[B]$, i.e.,

$$\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$$

Example 9 *Tossing a coin. The probability that heads shows up on two consecutive coin tosses,*

$$\Pr[HH] = \Pr[H].\Pr[H] = \frac{1}{2}.\frac{1}{2} = 0.25$$

Each toss of a coin is an Independent Event.

Now let's look at example where the individual samples are random, but the joint distributions are not.

Example 10 *Consider the following random variables X and Y both taking values in $\{0,1\}$ with the following joint distribution:*

$$\Pr[X=0 \wedge Y=0] = \frac{3}{16}, \quad \Pr[X=0 \wedge Y=1] = \frac{5}{16}$$
$$\Pr[X=1 \wedge Y=0] = \frac{5}{16}, \quad \Pr[X=1 \wedge Y=1] = \frac{3}{16}$$

Using the law of total probability, we have

$$\Pr[X=0] = \Pr[X=1] = \frac{1}{2}$$
$$\Pr[Y=0] = \Pr[Y=1] = \frac{1}{2}$$

but for each $x, y \in \{0,1\}$ we have

$$\Pr[X=a \wedge Y=b] \neq \Pr[X=a] \cdot \Pr[X=b]\,.$$

It is going to be important to keep this point in mind: even if individual distributions are uniform joint distributions need not be.

Note that we're using the symbol $\wedge$ indicating 'AND' to indicate independence while we've previously spoken about the set intersection $\cap$ when describing independence. These two symbols will be used to convey the same meaning; while $\cap$ will be used exclusively for sets, $\wedge$ will be used when we talk about random variables.

Pairwise Independent Random Variables: Let $X_1, X_2 \ldots\ldots X_n$ be random variables. We say that the X_i's are pairwise-independent if for every $i \neq j$ and all a and b, it holds that:

$$\Pr[X_i=a \wedge X_j=b] = \Pr[X_i=a] \cdot \Pr[X_j=a] \tag{2.4}$$

Example 11 *We throw two dice. Let (i) A be the event "the sum of the points is 7"; (ii) B the event "die #1 came up 3"; and (iii) C the event "die #2 came up 4".*

$$\Pr[A] = \Pr[B] = \Pr[C] = \frac{1}{6}$$

$$\Pr[A \cap B] = \Pr[B \cap C] = \Pr[A \cap C] = \frac{1}{36}$$

But,

$$\Pr[A \cap B \cap C] = \frac{1}{36} \neq \Pr[A] \cdot \Pr[B] \cdot \Pr[C]$$

A, B and C are pairwise independent but not independent as a triplet.

2.6 Tail Bounds

Markov's Inequality : Let X be a *non-negative random variable* and let $k \geq 1$. Then,

$$\Pr[X \geq k] \leq \frac{\mathbb{E}[X]}{k} \tag{2.5}$$

or

$$\Pr[X \geq k \cdot \mathbb{E}[X]] \leq \frac{1}{k} \tag{2.6}$$

Example 12 *Suppose we roll a single fair die and let X be the outcome. Then,*

$$\mathbb{E}[X] = 3.5$$
$$\mathsf{Var}[X] = \frac{35}{12}$$

Suppose we want to compute $\Pr[X \geq 6]$. We can easily see that $\Pr[X \geq 6] \geq \Pr[X = 6] \approx 0.167$. Using Markov's Inequality we can get an upper bound,

$$\Pr[X \geq 6] \leq \frac{3.5}{6} \approx 0.583$$

Markov's inequality gives an answer to the question "what is the probability that the value of the r.v., X, is 'far' from its expectation?".

Chebyshev's Inequality: Let X be a random variable, and $k \geq 1$. Then,

$$\Pr[|X - \mathbb{E}[X]| \geq k] \leq \frac{\mathsf{Var}[X]}{k^2} \tag{2.7}$$

Another answer to the question of "what is the probability that the value of X is far from its expectation" is given by Chebyshev's Inequality, which works for *any random variable* (not necessarily a non-negative one).

Example 13 *Let X be as defined in the above example. We can get a better bound on $\Pr[X \geq 6]$ using Chebyshev's inequality,*

$$\begin{aligned}\Pr[X \geq 6] &\leq \Pr[X \geq 6 \vee X \leq 1] \\ &= \Pr[|X - 3.5| \geq 2.5] \\ &\leq \frac{35}{12} \cdot \frac{1}{(2.5)^2} = \frac{7}{15} \approx 0.46\end{aligned}$$

Chernoff Bound: Let $X_1, X_2 \ldots X_n$ be independent random variables with $0 \leq X_i \leq 1$ (they need not have the same distribution). Let $X = X_1 + \ldots + X_n$, and $\mu = \mathbb{E}[X] = \mathbb{E}[X_1] + \cdots + \mathbb{E}[X_n]$, then for any $\delta \geq 0$

$$\Pr[X \geq (1+\delta)\mu] \leq e^{-\frac{\delta^2}{2+\delta}\mu} \tag{2.8}$$

This inequality is a particularly useful for analysis of the error probability of approximation via repeated sampling.

Example 14 *A biased coin, which lands heads with probability $\frac{1}{10}$ each time it is flipped, is flipped 200 times consecutively. We want an upper bound on the probability that it lands heads at least 120 times. The number of heads is a binomially distributed r.v., X, with parameters $p = \frac{1}{10}$ and $n = 200$. Thus, the expected number of heads is*

$$\mathbb{E}[x] = n \cdot p = 200 \cdot \frac{1}{10} = 20$$

Using Markov's inequality

$$\Pr[X \geq 120] \leq \frac{20}{120} = \frac{1}{6}$$

Using Chernoff bound we get,

$$\Pr[X \geq 120] = \Pr[X \geq (1+5) \cdot 20] \leq e^{-\frac{5^2}{2+5}20} \approx e^{-71.4}$$

which is a much better bound.

For specific random variables, particularly those that arise as sums of many independent random variables, we can get much better bounds on the probability of deviation from expectation. See Figure 2.8 for an example of how quickly the above example centers around the mean with a Gaussian distribution.

2.7 Model of Computation - Turing Machines

Throughout this course, it is important to understand the model of computation that we shall work with. This will be useful when we want to model various participants in the primitives and protocols we develop.

In this course, we shall limit our model of computation to that of Turing Machines, which we describe informally below. While it is not important to understand the workings of a Turing Machine, it is presented here for completeness. The below description is taken verbatim from [Kat].

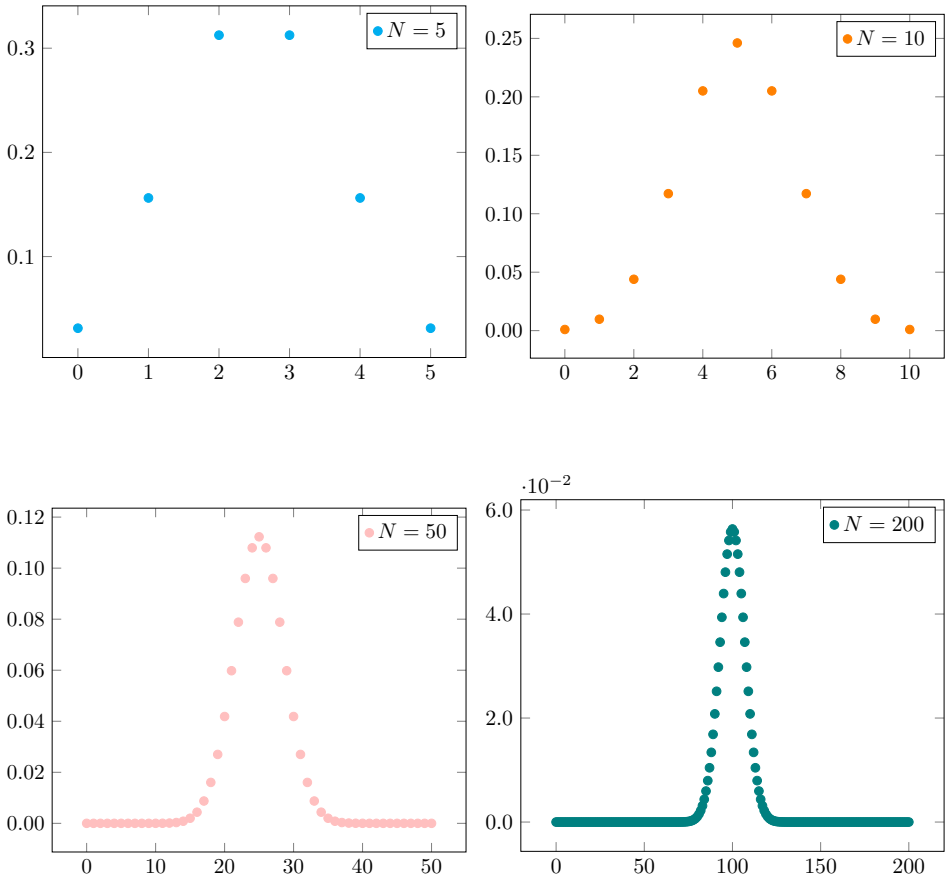


Figure 2.8: The above example, taken from [Bar] illustrates the Chernoff bound shows how the sum of coin tosses quickly converge (with N, the total number of samples) to the Gaussian Distribution; where the probability density is centered around the mean.

A Turing Machine is defined by an integer $k \geq 1$, a finite set of states Q, an alphabet Γ, a transition function $\delta : Q \times \Gamma^k \mapsto Q \times \Gamma^{k-1} \times \{\mathsf{L}, \mathsf{S}, \mathsf{R}\}^k$ where:

- k is the number of infinite, one-dimensional tapes used by the machine. We typically have $k \geq 3$, where the first tape is denoted as the the *input tape*, and the last tape the *output tape*. The position of the tape currently being read is specified by a separate tape head for each tape.

- the set of states Q contains two special states: (a) the start state q_{start}; and (b) the halt state q_{halt}.
- Γ contains $\{0, 1\}$, a special "blank symbol", and a special "start symbol".

The computation of a Turing machine M on input $x \in \{0,1\}^*$ proceeds as follows: All tapes of the Turing machine contain the start symbol followed by the blank symbols, with the exception of the input tape which contains the input x. The machine starts in state $q = q_{\mathsf{start}}$ with its k heads at the leftmost position of each tape. Then, until q is the halt state, repeat the following:

1. Let the current contents of the cells being scanned by the k heads be $\gamma_1, \cdots, \gamma_k \in \Gamma$.
2. Compute $\delta(q, \gamma_1, \cdots, \gamma_k) = (q', \gamma'_2, \cdots, \gamma'_k, D_1, \cdots, D_k)$ where $q' \in Q$, $\gamma'_2, \cdots, \gamma'_k \in \Gamma$ and $D_i \in \{\mathsf{L}, \mathsf{S}, \mathsf{R}\}$.
3. Overwrite the contents of the currently scanned cell on tape i to γ'_i for $2 \leq i \leq k$; move head i to the left, to the same position, or to the right depending on whether D_i = L, S, or R, respectively; and then set the current state to $q = q'$.

The output of M on input x, denoted $\mathsf{M}(x)$, is the binary string contained on the output tape when the machine halts. It is possible that M never halts for certain inputs x.

The above description is that of a **Deterministic Turing Machine**. We contrast this with a **Non-Deterministic Turing Machine** (NDTM). An NDTM instead of having a single transition function δ, has *two* transition function δ_0, δ_1 and at each step decides in a non-deterministic/arbitrary manner. Therefore, after n steps the machine can be in at most 2^n configurations, where a configuration is the joint description of all the tapes of the Turing Machine.

We fill often find it more convenient to talk about algorithms, defined below.

Definition 4 (Algorithm) *An algorithm is a Turing Machine whose input and output are strings over the binary alphabet $\Sigma = \{0, 1\}$.*

Note, that the two terms *Turing Machine* and *Algorithm* will be used interchangeably from now on.

Definition 5 (Running Time) *An algorithm A is said to run in time $T(n)$ if for all strings of length n over the input alphabet ($x \in \{0,1\}^n$), $A(x)$ halts within $T(|x|)$ steps.*

In this course, we will primarily focus on algorithms that have a *polynomial* running time.

Definition 6 (Polynomial Running Time) *An algorithm A is said to run in polynomial time if there exists a constant c such that A runs in time $T(n) = n^c$.*

We say an algorithm is *efficient* if it runs in polynomial time. Even for efficient algorithms, the constant c can be a large value. For example, consider c=100. In practice, n^{100} may actually be considered "inefficient". For our purposes, however, we will stick with this definition of efficiency.

If an algorithm runs exponential or super-polynomial time, i.e., $T(n) = 2^n$ or $T(n) = n^{(\log n)}$, then we will say it is *inefficient.*

One might consider other notions of efficiency, and there is nothing pristine about using polynomial time as the measure of efficiency. But throughout this course, we shall see that this is often a convenient measure since the composition of polynomials remain a polynomial.

So far, we have only considered deterministic algorithms. In computer science, and specifically cryptography, randomness plays a central role. Therefore, throughout the course, we will be interested in randomized (a.k.a. probabilistic) algorithms.

Definition 7 (Randomized Algorithm) *A randomized algorithm, also called a probabilistic polynomial time Turing machine (*PPT*) is a Turing machine that runs in polynomial time and is equipped with an extra randomness tape. Each bit of randomness tape is uniformly and independently chosen.The output of a randomized algorithm is a distribution.*

Think about the difference between a NDTM and a Randomized Algorithm.

Remark 2 *Note that we do not place any limits on the length of the ranom tape, once the randmness has been fixed, the computation is completely determinisitic.*

When we talk about randomized algorithms, wherever possible we shall make explicit the randomness used. So an algorithm $\mathsf{M}(x; r)$ indicates that M runs on input x using randomness r that is sampled as $r \leftarrow\!\!\$\ \{0,1\}^m$ for some m. In fact, one can think of the output of $\mathsf{M}(x; r)$ as a random variable with randomness from r.

As mentioned earlier, in practice, everyone including the adversary has some bounded computational resources. These resources can be used in a variety of intelligent ways, but they are still limited. Turing machines are able to capture all the types of computations possible given these resources. Therefore, a adversary will be a computer program or algorithm modeled as a Turing machine.

This captures what we can do efficiently ourselves and can be described as a uniform PPT Turing machine. When it comes to adversaries, we will allow them

to have some extra power. Instead of having only one algorithm that works for different input lengths, it can write down potentially a different algorithm for every input size. Each of them individually could be efficient. If that is the case, overall the adversary still runs in polynomial time.

The notion of non-uniform TMs has a connection with another model of computation, computation by circuits. Look up the connection.

Definition 8 (Non-Uniform PPT Machine) *A non-uniform probabilistic polynomial time Turing machine is a Turing machine A made up of a sequence of probabilistic machines $\mathsf{M} = \{\mathsf{M}_1, \mathsf{M}_2, \cdots\}$ for which there exists a polynomial $\mathsf{p}(\cdot)$ such that for every $\mathsf{M}_i \in \mathsf{M}$, the description size $|\mathsf{M}_i|$ and the running time of M_i are at most $\mathsf{p}(i)$. We write $\mathsf{M}(x)$ to denote the distribution obtained by running $\mathsf{M}_{|x|}(x)$.*

Our adversary will usually be a non-uniform probabilistic polynomial running time algorithm (n.u. PPT).

2.8 Complexity Classes

We'll discuss some important complexity classes now. Each of these complexity classes is a collection of languages that share some common property. A language is simply a subset of $\{0,1\}^*$.

We say Turing Machine M *decides* a language L if $x \in L \implies \mathsf{M}(x) = 1$, and $x \notin L \implies \mathsf{M}(x) = 0$. We can now define the complexity classes.

Complexity Class P: A language L is in P if there exists a (deterministic) Turing Machine M_L and a polynomial $\mathsf{p}(\cdot)$ such that:

- on input strings x, machine M halts after at most $\mathsf{p}(|x|)$ steps; and
- M_L decides L.

Complexity Class NP: There are two (equivalent) ways to define the class NP. The classical way is with respect to NDTMs. We say that an NDTM M outputs 1 if there is *at least* one sequence of non-deterministic choices that results in the output tape containing 1. Similar to P, a language L is in NP if there a polynomial time NDTM M_L that decides L.

For us, it will be convenient to work with the alternate definition because it is defined with respect to : A language L is in NP if there exists a Boolean relation $R_L \subseteq \{0,1\}^* \times \{0,1\}^*$ and a polynomial $\mathsf{p}(\cdot)$ such that R_L can be recognized in (deterministic) polynomial time, and $x \in L$ if and only if there exists a y such that $|w| \leq (\mathsf{p}|x|)$ and $(x, w) \in R_L$. Such a w is called a witness for membership of $x \in L$.

Try to think of why the two notions are equivalent.

Polynomial Reduction: While this is not a class, we will need it to define the subsequent two classes. We shall not provide a formal treatment here. The notion of a reduction is useful for answering questions of the form "Is language L easier that language L'?".

> We say a language L is reducible to a language L' if there exists a polynomial-time computable function f such that $x \in L$ if and only if $f(x) \in L'$. This is expressed by writing $L \leq_p L'$.

This is called a *Karp reduction*. There are other notions of reductions in computer science.

This lets us state that L is no harder than L' since any Turin Machine M that decides L' can be used to decide L.

NP-Hardness: A language is NP−Hard if every language in NP is polynomially reducible to it.

Thus NP−Hard is the set of all languages that are at least as hard as any problem in NP.

NP-Completeness: A language is NP−Complete if it is both (i) in NP; and (ii) in NP−Hard.

The class NP−Complete consists the set of languages (or problems) that represent all of NP on account of being the hardest problems in NP. So any statement we want to make about all of NP, it suffices to make about any language that is NP−Complete.

Our current known understanding of the relationship between the complexity classes discussed is illustrated in Figure 2.9.

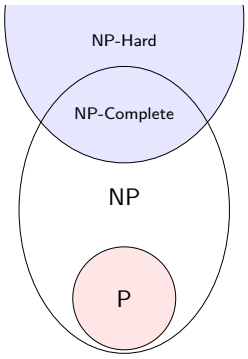


Figure 2.9: Relation between common complexity classes

Bounded error Probabilistic Polynomial Time, BPP**:** We now define a language for randomized Turing Machines. We say that L is recognized by the probabilistic polynomial Time Turing machine M if:

- for $x \in L$, then $\Pr[\mathsf{M}(x) = 1] \geq 2/3$
- for $x \notin L$, then $\Pr[\mathsf{M}(x) = 1] \leq 1/3$

2.9 Asymptotic Notations

When describing the running time of algorithms, instead of describing the running as a complicated function in the size of the input, it is more convenient to describe the running time with the function that reflects the growth. In view of this, we shall see that for large inputs, the multiplicative factors and lower order terms are dominated by the effects of the input size.

Big-O. The function $f(n)$ is $O(g(n))$ if $\exists$ constants c, n_0, such that $\forall n \geq n_0$

$$0 \leq f(n) \leq c \cdot g(n)$$

Big-Omega. The function $f(n)$ is $\Omega(g(n))$ if $\exists$ constants c, n_0, such that $\forall n \geq n_0$

$$0 \leq c \cdot g(n) \leq f(n)$$

Theta. The function $f(n)$ is $\theta(g(n))$ if $\exists$ constants c_1, c_2, n_0, such that $\forall n \geq n_0$

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

We have illustrated these notions, and the importance of the constants c and n_0, in Figure 2.10.

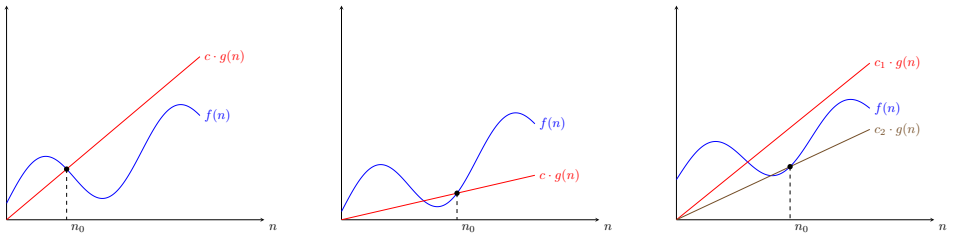


Figure 2.10: Here we give examples of the O, Ω and θ notations.

The above asymptotic notations of O and Ω may not be asymptotically, so below we describe their corresponding asymptotically tight notions.

Small-o. The function is $f(n)$ is $o(g(n))$ if $\forall$ constant c, $\exists$ constant n_0, such that $\forall$

$$0 \leq f(n) < c \cdot g(n)$$

Small-omega. The function is $f(n)$ is $\omega(g(n))$ if $\forall$ constant c, $\exists$ constant n_0, such that $\forall$

$$0 \leq c \cdot g(n) < f(n)$$

Note how the quantifiers, and their order, have changed in the above definition. Recall from our discussion of quantifiers how the order affects the definition.

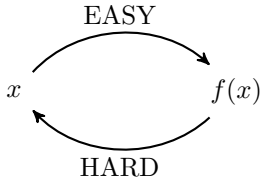
Remark 3 *We have slightly abused notation above since $O(g(n))$ defines sets, but we shall find it convenient to use the above description. See Chapter 3 of [CLRS09] for a detailed discussion of these concepts.*

Chapter 3

One-Way Functions

3.1 Introduction

Intuitively, a given function f is "one-way" if it is very easy to compute $f(x)$ efficiently, but it is hard to recover x if given $f(x)$. Over the course, we shall see importance of one-way function, and its role in cryptography. Before we can get there, we need to formally define our above intuition for a function f to be one-way. Let's take a stab at such a definition below.



Definition 9 (One-way Function (Informal)) *A function f is one-way if it satisfies the following two informal properties:*

1. ***Functionality - Easy to compute:*** *Given any input x from the domain, it should be easy to compute $f(x)$. In other words it is possible to compute $f(x)$ in polynomial time.*
2. ***Security - Hard to invert:*** *Any polynomial time algorithm should fail to recover x given $f(x)$. This can also be expressed as: the probability of inverting $f(x)$ is "small".*

Who's trying to invert this function? We'll always assign this task to an adversary who we shall try to thwart. Let's make a couple of notes about the adversary.

Adversary's Resources: In practice, everyone has bounded computational resources. Therefore, it is reasonable to model the adversary as such an entity, instead of an all powerful entity. But based on such a choice, we arrive at two different notions of security:

Computational Security: Security against efficient adversaries. We will mostly focus on computational security throughout the course.
Information-theoretic Security: Security against inefficient adversaries.

Adversary's Strategy: The adversary is not restricted to any specific strategies. We do not make any assumptions about adversarial strategy. Adversary can use its bounded computational resources however intelligently it likes.

Turing Machines can capture all types of computations that are possible. Hence, our adversary will be a computer program or an algorithm, modeled as a Turing Machine (see Section 2.7). Our adversary will also be efficient (captured via its running time). We will give the adversary additional power of randomness and non-uniformity and consider adversaries to by non-uniform PPT Turing Machines.

When you see a definition that talks about probability, always ask yourself what the probability is over and where the randomness in the probability comes from.

Notice above that we said the *probability of inverting $f(x)$ is small.* What is this probability even over? We've talked about modeling our adversaries as PPT machines with access to a random tape. So a natural idea would be to consider the probability over the adversary's random tape. But if you think about it a little, this isn't a good idea. A non-uniform adversary can find the best available random tape and just hardwire that into the description.

To avoid the adversary having control over the randomness, we instead take the probability over the choice of x. Now the intuition for f to be a one-way function is the following: take any non-uniform PPT adversary, with the best possible strategy, the probability that $\mathcal{A}$ inverts $f(x)$ for a randomly chosen x from the input should be small. So we sample an x at random, and then we compute $f(x)$ and give it to the adversary, and we observe the probability that it inverts that image.

Using this informal description, we will make an attempt to describe the 2 conditions in a more formal way.

3.2 Formal Definition of One-way Functions

With the intuition described in the previous section, let's write down our first attempt at a one-way function.

Definition 10 (One-way Function (1^{st} Attempt)) *A function $f : \{0,1\}^* \mapsto \{0,1\}^*$ is a one way function (OWF) if it satisfies the following two conditions:*

1. **Easy to compute***: There is a PPT algorithm C s.t. $\forall x \in \{0,1\}^*$,*

$$\Pr[r \leftarrow_\$ \{0,1\}^m \ : \ C(x;r) = f(x)] = 1,$$

   where the probability of success for C to output the correct value $f(x)$ on input x, is taken over the random coins used by C to compute $f(x)$. For simplicity, we denote here the number of random coins by m.
2. **Hard to invert***: For every non-uniform PPT adversary $\mathcal{A}$, for any input length $n \in \mathbb{N}$.*

$$\Pr[x \leftarrow_\$ \{0,1\}^n : \mathcal{A} \text{ inverts } f(x)] \leq \text{small}$$

We need to specify what exactly "small" means. To this end, we will define a fast decaying function $\nu(\cdot)$ s.t for any imput length $n \in \mathbb{N}$ this function decays asymptotically faster than any inverse polynomial. We will call this function *negligible*. Let us formalize this below.

Definition 11 (Negligible function) *A function $\nu(\cdot)$ is negligible if for $\forall c \in \mathbb{N}$, $\exists n_0 \in \mathbb{N}$, such that $\forall n \geq n_0$,*

$$\nu(n) < \frac{1}{n^c}$$

As we stated earlier, a negligible function decays faster than all "inverse-polynomial" functions $\left(n^{-\omega(1)}\right)$. Examples of an obviously negligible function are exponentially decaying functions 2^{-n} or $n^{-\log(n)}$.

Updating our previous definition of one-way functions in view of our definition of negligible functions, we have our second attempt.

Definition 12 (One-way Function (2^{nd} Attempt)) *A function $f : \{0,1\}^* \mapsto \{0,1\}^*$ is a one way function (OWF) if it satisfies the following two conditions:*

1. **Easy to compute***: There is a PPT algorithm C s.t. $\forall x \in \{0,1\}^*$,*

$$\Pr[r \leftarrow_\$ \{0,1\}^m \ : \ C(x;r) = f(x)] = 1.$$

2. **Hard to invert***: For every non-uniform PPT adversary $\mathcal{A}$, for any input length $n \in \mathbb{N}$, there exists a negligible function $\nu(\cdot)$ such that*

$$\Pr[x \leftarrow_\$ \{0,1\}^n : \mathcal{A} \text{ inverts } f(x)] \leq \nu(|x|)$$

Although this definition seems accurate, it has one small problem: What is $\mathcal{A}$'s input? Let's take a closer look at this, and let $y := f(x)$. If f is a one way function, the following two conditions have to be satisfied by $\mathcal{A}$:

- **Condition 1**: $\mathcal{A}$ on input y must run in $\mathsf{poly}(|y|)$.
- **Condition 2**: $\mathcal{A}$ cannot output x' s.t. $f(x') = y$.

However, if the size of y is much smaller than the size of the domain, $\mathcal{A}$ cannot write the inverse even if it can find it. For example, consider the function

$$f(x) = \text{first } \log|x| \text{ bits of } x.$$

Although it is trivial to invert this function,

$$f^{-1}(y) = y||\underbrace{0000...0}_{n-\log n}$$

where $n = 2^{|y|}$, it still satisfies the definition for one-way function we've proposed. Let's see why this is true. Although f is easy to compute, $\mathcal{A}$ cannot invert f in time $\mathsf{poly}(|y|)$ as it needs $2^{|y|}$ to write down the answer. But this clearly a function we want to rule out as a one-way function. In order to fix this issue, we adopt the convention to always pad y and write $\mathcal{A}(1^n, y)$, so that $\mathcal{A}$ has sufficient time to write the answer.

Putting this all together we have the following final definition.

Definition 13 (One-way Function) *A function $f : \{0,1\}^* \mapsto \{0,1\}^*$ is a one way function (OWF) if it satisfies the following two conditions:*

1. **Easy to compute***: There is a PPT algorithm C s.t. $\forall x \in \{0,1\}^*$,*

$$\Pr[r \leftarrow_\$ \{0,1\}^m \ : \ C(x;r) = f(x)] = 1.$$

2. **Hard to invert***: For every non-uniform PPT adversary $\mathcal{A}$, for any input length $n \in \mathbb{N}$, there exists a negligible function $\nu(\cdot)$ such that*

$$\Pr[x \leftarrow_\$ \{0,1\}^n : \widetilde{x} \leftarrow \mathcal{A}(1^n, f(x)) : f(\widetilde{x}) = f(x)] \leq \nu(n)$$

As we shall see, it is instructive to represent the above definition in terms of game between a challenger, and an adversary.

From the above diagram it is clear that f is a one-way function, if for every adversary $\mathcal{A}$, the challenger outputs 1 with only negligible probability $\nu(n)$. When clear from the context, we shall avoid clutter sometimes and drop the 1^n in the challenger-adversary diagrams.

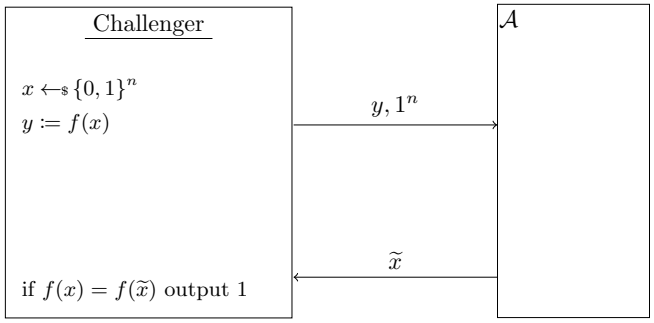


Figure 3.1: One-way function challenge game

An important exercise to write out what it means for a function f to *not* be a one-way function. Specifically, we say that a function f is not a one-way function if there exists an adversary $\mathcal{A}$, a polynomial q such that for infinitely many values of n,

$$\Pr[x \leftarrow\!\!\$\ \{0,1\}^n : \widetilde{x} \leftarrow \mathcal{A}(1^n, f(x)) : f(\widetilde{x}) = f(x)] \geq \frac{1}{\mathsf{q}(n)}.$$

Remark 4 *Before we proceed, let's stop to look at the above phrasing which talks about* infinitely *many n. Why did we choose to phrase it as such. This goes back to our definition of negligible functions, for which we said must hold for all n greater than some n_0. If instead of infinitely many n, there were only a large, but finitely many n, we could always pick n_0 to be largest value in this set, and then this would not contradict the definition of a one-way function. So for a function to not be one-way, it must be the case that the above holds for infinitely many n.*

Extensions. In the chapter discussing preliminaries, we've already talked about injective or 1-1 functions, and permutations. The above definition for one-way functions extend naturally when the function f is additionally restricted to be an injective function, or a permutation.

3.3 Factoring Problem

Existence of one-way functions. Given that we're discussing one-way function, we should stop to ask if these objects even exist. It turns out that they don't exist unconditionally, and at the very least requires proving $\mathsf{P} \neq \mathsf{NP}$. However, by making certain assumptions about the hardness of some problems, we can construct conditional one-way functions also called "candidates". One such problem whose hardness is well studied is the Factoring Problem.

We will start with considering the **multiplication** function: $f_x : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$:

$$f_x(x, y) = \begin{cases} \perp & x = 1 \text{ or } y = 1 \\ x \cdot y & \text{otherwise} \end{cases}$$

This function is clearly not one-way, as the probability of xy being even and thus obviously factored into $(2, xy/2)$ is $\frac{3}{4}$ for random (x, y). In other words, the inversion succeeds 75% of time. We will then try to eliminate such trivial factors. We define $\prod_n$ be the set of all prime numbers $< 2^n$ and we randomly select two elements, p and q from this set and multiply them. Their product is thus unlikely to contain small trivial factors. Let's state our assumption formally below.

Assumption 1 (Factoring Assumption) *For every (non-uniform PPT) adversary $\mathcal{A}$, there exists a negligible function ν such that:*

$$\Pr\left[p \leftarrow_{\$} \Pi_n; q \leftarrow_{\$} \Pi_n; N \coloneqq pq \ : \ \mathcal{A}(N) \in \{p, q\}\right] \leq \nu(n)$$

Of course, when we state an assumption, we must ask if it is a reasonable assumption. For the factoring assumption, up until now, there have been no "good attacks". The best known algorithms for breaking the Factoring Assumption are:

$$2^{O(\sqrt{n \log n})} \qquad \text{(provable)}$$

$$2^{O(\sqrt[3]{n \log^2 n})} \qquad \text{(heurestic)}$$

Looking back at our multiplication function, it is clear that if a random x and y happen to be prime, no $\mathcal{A}$ could invert it, which is the *good* case. If such a case happens with probability greater than ϵ, then every $\mathcal{A}$ must fail to invert the function with probability at least ϵ. If ϵ is a noticeable function, then $\mathcal{A}$ fails to invert the function with noticeable probability. This doesn't satisfy our requirement for a one-way function, but let us relax the notion slightly. This is what we shall call a *weak* one-way function. Before we define a *weak* one-way function, let us start by defining a noticeable function.

Definition 14 (Noticeable Function) *A function $\epsilon : \mathbb{N} \mapsto \mathbb{R}$ is noticeable if there exits $c \in \mathbb{N}$ and $n_0 \in \mathbb{N}$ such that $\forall n > n_0$:*

$$\epsilon(n) \geq \frac{1}{n^c}$$

Given the above definition, we have the following definition for a weak one-way function.

Definition 15 (Weak One-way Function) *A function* $f : \{0,1\}^* \mapsto \{0,1\}^*$ *is a one way function (OWF) if it satisfies the following two conditions:*

1. **Easy to compute***: There is a PPT algorithm* C *s.t.* $\forall x \in \{0,1\}^*$,

$$\Pr[r \leftarrow_\$ \{0,1\}^m \ : \ C(x;r) = f(x)] = 1.$$

2. **Somewhat hard to invert***: There is a noticeable function* $\epsilon : \mathbb{N} \mapsto \mathbb{R}$ *s.t. for every non-uniform PPT adversary* $\mathcal{A}$*, for any input length* $n \in \mathbb{N}$,

$$\Pr[x \leftarrow_\$ \{0,1\}^n : \widetilde{x} \leftarrow \mathcal{A}(1^n, f(x)) : f(\widetilde{x}) \neq f(x)] \geq \epsilon(n).$$

The reader should note the differences between this definition and our earlier definition of a *strong* one-way function.

Now we will try to show that $f_\times$ is indeed a weak OWF.

Theorem 2 *Assuming the factoring assumption, function* $f_\times$ *is a weak OWF.*

To prove this, we will show that the "good" case when x and y are prime occurs with noticeable probability, and to this end we will use Chebyshev's theorem to show that the fraction of prime numbers between 1 and 2^n is noticeable.

Theorem 3 (Chebyshev's theorem) *An* n *bit number is a prime with probability* $\frac{1}{2n}$.

Now we proceed to the proof.

Proof We're going to prove this using two different approaches. Our latter approach will be the more common approach of proof by contradiction.

Proof via definition : Let GOOD be the set of inputs (x, y), such that both x and y are prime. Then we have

$$\begin{aligned}\Pr[\mathcal{A} \text{ inverts } f_\times] = \Pr[\mathcal{A} \text{ inverts } f_\times \mid (x,y) \in \mathsf{GOOD}] \cdot \Pr[(x,y) \in \mathsf{GOOD}] \\ + \Pr[\mathcal{A} \text{ inverts } f_\times \mid (x,y) \notin \mathsf{GOOD}] \cdot \Pr[(x,y) \notin \mathsf{GOOD}]\end{aligned}$$

According to the Factoring Assumption, when $(x, y) \in \mathsf{GOOD}$, $\mathcal{A}$ could invert $f_\times$ with a probability no more than a negligible function $\nu(n)$. Using Chebyshev's

theorem, an n bit number is a prime number with probability $\frac{1}{2n}$. Thus we get

$$\Pr[\mathcal{A} \text{ inverts } f_\times] \leq \nu(n) \cdot \frac{1}{4n^2} + 1 \cdot \left(1 - \frac{1}{4n^2}\right) = 1 - \frac{1}{4n^2}(1 - \nu(n))$$

Now we only need to prove that $\frac{1}{4n^2}(1-\nu(n))$ is a noticeable function. Given that $\forall c > 0$,$\nu(n) \leq \frac{1}{n^c}$, we can conclude that for $n \geq 2$, $1 - \nu(n) \geq \frac{1}{n}$. Thus $\frac{1}{4n^2}(1-\nu(n)) \geq \frac{1}{4n^3}$ is noticeable. Hence $f_\times$ is a weak OWF.

Now, let us prove the same statement via reduction to the factoring assumption.

Proof via reduction: Suppose that $f_\times$ is not a weak OWF, then we can construct an adversary to break the factoring assumption. Assume that there exists a non-uniform PPT algorithm $\mathcal{A}$ inverting $f_\times$ with probability at least $1 - \frac{1}{8n^2}$. That is

$$\Pr\left[(x,y) \leftarrow_\$ \{0,1\}^n \times \{0,1\}^n, z \coloneqq x \cdot y \ : \ \mathcal{A}(1^{2n}, z) \in f_\times^{-1}(z)\right] \geq 1 - \frac{1}{8n^2}.$$

Now we construct a non-uniform adversary algorithm $\mathcal{B}$, which on input z (which is a product of two random n-bit prime numbers) uses $\mathcal{A}$ to break the factoring assumption. This is depicted pictorially below.

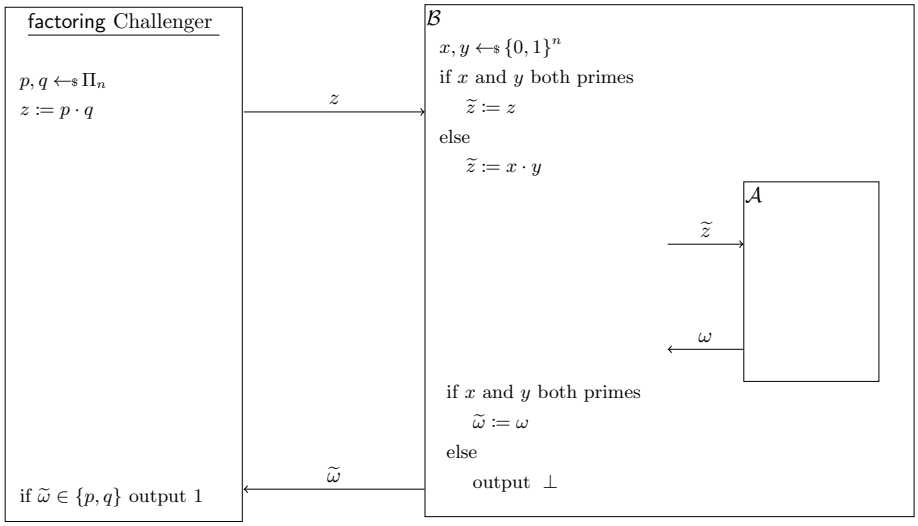


Figure 3.2: Reduction to factoring assumption

Since this is our first proof by reduction, we've also written down the steps separately. In the future, we shall only use the above pictorial representation.

$\mathcal{B}(z)$

1 : Pick (x, y) randomly from $\{0,1\}^n \times \{0,1\}^n$;
2 : if x, y are both prime, let $\widetilde{z} = z$;
3 : else, let $\widetilde{z} = xy$;
4 : run $\omega = A(1^{2n}, \widetilde{z})$;
5 : if x, y are both prime, set $\widetilde{\omega} \coloneqq \omega$, and return $\widetilde{\omega}$.
6 : else output $\perp$ to indicate failure.

Remark 5 *The reason for randomly choosing (x, y) instead of passing the input directly to $\mathcal{A}$ is that, the input of $\mathcal{B}$ is a product of two random n-bit primes while that of $\mathcal{A}$ is the product of two random n-bit numbers. Passing the input directly to $\mathcal{A}$ would not emulate the uniform distribution of the inputs given to $\mathcal{A}$.*

Now we calculate the probability that $\mathcal{B}$ fails to break factoring assumption. We use the following notation:

$$\begin{aligned}\Pr[\mathcal{B} \text{ fails}] &= \Pr[\mathcal{B} \text{ passes input to } \mathcal{A}] \cdot \Pr[\mathcal{A} \text{ fails to invert } f_\times] \\ &\quad + \Pr[\mathcal{B} \text{ fails to pass input to } \mathcal{A}] \\ &\leq \Pr[\mathcal{A} \text{ fails to invert } f_\times] + \Pr[\mathcal{B} \text{ fails to pass input to } \mathcal{A}] \\ &\leq \frac{1}{8n^2} + \left(1 - \frac{1}{4n^2}\right) \leq 1 - \frac{1}{8n^2}\end{aligned}$$

Thus $\mathcal{B}$ breaks factoring assumption with a noticeable probability. And we get contraction. Thus $f_\times$ is a weak one-way function. □

3.4 Weak to strong One-way function

Once we have a weak one-way function, how do we get a strong one-way function? Can we transform the multiply function into a strong OWF? Can we do this generically? The answer is yes, and was proven by Yao.

Theorem 4 (Yao's Theorem) *Strong OWFs exist if and only if weak OWFs exist.*

In other words, if you have a weak one-way function, you can generically convert it to a strong one-way function.This is an example of a general phenomenon which is very well studied in complexity theory called *hardness amplification.*

Theorem 5 *For any weak OWF $f : \{0,1\}^n \mapsto \{0,1\}^n$, $\exists$ polynomial $N(\cdot)$ s.t. $F : \{0,1\}^{nN(n)} \mapsto \{0,1\}^{nN(n)}$ defined as*

$$F(x_1, ..., x_{N(n)}) = \Big(f(x_1), ..., f(x_{N(n)})\Big)$$

is a strong OWF.

Proof Since f is weak OWF, from Definition 15 we have $\mathsf{q} : \mathbb{N} \mapsto \mathbb{N}$ to be a polynomial function, such that for every non-uniform $\mathcal{A}$,

$$\Pr[x \leftarrow_{\$} \{0,1\}^n, y \coloneqq f(x) \ : \ f\left(\mathcal{A}(1^n, y)\right) = y] \leq 1 - \frac{1}{\mathsf{q}(n)}.$$

We want to find a N the right hand side of the above equation when repeatedly applied yields a negligible function. Specifically, we want an N such that $\left(1 - \frac{1}{\mathsf{q}(n)}\right)^N$ is negligible. One such value of N is $N \coloneqq 2n\mathsf{q}(n)$ which gives

$$\left(1 - \frac{1}{q(n)}\right)^N \approx \frac{1}{e^{2n}}.$$

Suppose that the defined F is not a strong OWF. Then by our definition of OWFs, $\exists$ polynomial function $\mathsf{p}'(\cdot)$ and a non-uniform adversary $\mathcal{A}$ s.t.

$$\begin{aligned}\Pr\Big[(x_1, ..., x_N) \leftarrow_{\$} \{0,1\}^{nN}; y \coloneqq F(x_1, ..., x_N) \ : \ F\Big(\mathcal{A}(1^{nN}, y)\Big) = y\Big] &\geq \frac{1}{\mathsf{p}'(nN)} \\ &= \frac{1}{\mathsf{p}(n)}\end{aligned}$$

This follows from the fact that N is a polynomial in n, therefore $\mathsf{p}'(nN)$ can be rewritten as a polynomial $\mathsf{p}(n)$ solely in n.

Now we shall use $\mathcal{A}$ to construct a non-uniform PPT $\mathcal{B}$ that breaks f with probability larger than $1 - \frac{1}{q(n)}$.

As a first step, we construct an intermediate adversary $\mathcal{B}_0$ on input $y = f(x)$ for random $x \in \{0,1\}^n$ as shown in Figure 3.3.

To improve the chance of inverting f, we construct $\mathcal{B}$ to run $\mathcal{B}_0$ several times using independently chosen random coins. We define $\mathcal{B} : \{0,1\}^n \mapsto \{0,1\}^n \cup \{\perp\}$ on input y to run $\mathcal{B}_0(y)$ for $2nN^2\mathsf{p}(n)$ times independently (i.e. choose x_j independently and randomly each time). $\mathcal{B}$ then outputs the first non-$\perp$ it receives. If all runs of $\mathcal{B}_0$ results in $\perp$, then $\mathcal{B}$ also outputs $\perp$.

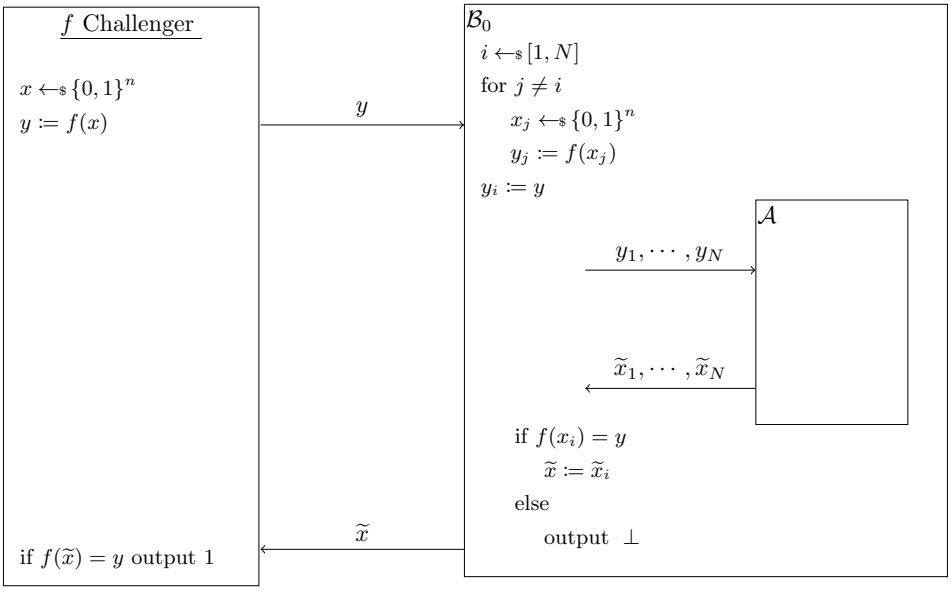


Figure 3.3: Construction of $\mathcal{B}_0$ from $\mathcal{A}$

Let GOOD be the set of xs that $\mathcal{B}_0$ inverts f with a probability at least $\frac{1}{2N^2\mathsf{p}(n)}$, i.e.,

$$\mathsf{GOOD} = \left\{ x \in \{0,1\}^n \;\middle|\; \Pr[f(\mathcal{B}_0(1^n, f(x))) = f(x)] \geq \frac{1}{2N^2\mathsf{p}(n)} \right\}.$$

Otherwise, we call x *bad*. Note that here the probability on the right side above is with respect to random coins used by $\mathcal{B}_0$.

The probability that $\mathcal{B}$ fails to invert f on $f(x)$ for x in GOOD is the probability that $\mathcal{B}$ fails on all $2nN^2\mathsf{p}(n)$ calls to $\mathcal{B}_0$, i.e.

$$\Pr[\mathcal{B}(f(x)) \text{ fails} \mid x \in \mathsf{GOOD}] \leq \left(1 - \frac{1}{2N^2\mathsf{p}(N)}\right)^{2nN^2\mathsf{p}(n)} \approx \frac{1}{e^n},$$

which is extremely small.
We prove that the fraction of GOOD set is noticeable.

Lemma 6 *There are at least* $2^n \cdot \left(1 - \frac{1}{2\mathsf{q}(n)}\right)$ *elements in* $\{0,1\}^n$ *in* GOOD.

Before we proceed to proving this Lemma, let's see how one would use it to complete the proof. We show that as long as the set GOOD is sufficiently large, $\mathcal{B}$ will

succeed with high probability over the choice of x, alternatively that probability that $\mathcal{B}$ fails is low.

$$\begin{aligned}
\Pr[\mathcal{B}(f(x)) \text{ fails}] &= \Pr[\mathcal{B}(f(x)) \text{ fails} \mid x \in \mathsf{GOOD}] \cdot \Pr[x \in \mathsf{GOOD}] \\
&\quad + \Pr[\mathcal{B}(f(x)) \text{ fails} \mid x \notin \mathsf{GOOD}] \cdot \Pr[x \notin \mathsf{GOOD}] \\
&\leq \Pr[\mathcal{B}(f(x)) \text{ fails} \mid x \in \mathsf{GOOD}] + \Pr[x \notin \mathsf{GOOD}] \\
&\leq \left(1 - \frac{1}{2N^2\mathsf{p}(n)}\right)^{2N^2 n\mathsf{p}(n)} + \frac{1}{2\mathsf{q}(n)} \\
&\approx e^{-n} + \frac{1}{2\mathsf{q}(n)} \\
&< \frac{1}{\mathsf{q}(n)}
\end{aligned}$$

The first inequality comes from upper bounding the probability by 1. The second inequality comes from Lemma Lemma 6 and our earlier calculation of $\mathcal{B}$ failing to invert $f(x)$ when $x \in \mathsf{GOOD}$.

This is a contradiction to our assumption that f is $\mathsf{q}(n)$- weak. The only thing that remains to be proven is Lemma 6.

Proof (lemma 6) For the sake of contradiction, assume there are $> 2^n \cdot \left(\frac{1}{2q(n)}\right)$ elements not in GOOD. We shall show this violates the assumption that $\mathcal{A}$ inverts F with probability at least $\frac{1}{\mathsf{p}(n)}$. For simplicity of notation assume that that $\mathcal{A}$ outputs the symbol $\perp$ when it does not succeed, else it outputs the correct inverse. Also, we ignore the input 1^{nN} to $\mathcal{A}$.

$$\begin{aligned}
&\Pr\left[(x_1, ..., x_N) \leftarrow_\$ \{0,1\}^{nN}; y \coloneqq F(x_1, ..., x_N) \ : \ \mathcal{A}(y) \neq \perp\right] \\
&= \Pr\left[(x_1, ..., x_N) \leftarrow_\$ \{0,1\}^{nN}; y \coloneqq F(x_1, ..., x_N) \ : \ \mathcal{A}(y) \neq \perp \wedge (\forall i, x_i \in \mathsf{GOOD})\right] \\
&+ \Pr\left[(x_1, ..., x_N) \leftarrow_\$ \{0,1\}^{nN}; y \coloneqq F(x_1, ..., x_N) \ : \ \mathcal{A}(y) \neq \perp \wedge (\exists i, x_i \notin \mathsf{GOOD})\right]
\end{aligned}$$

For the first term, it suffices to bound the probability that *all* x_i are in GOOD. This is because $\mathcal{A}$ can at best invert with probability 1 in this case.

$$\begin{aligned}
&\Pr\left[(x_1, ..., x_N) \leftarrow_\$ \{0,1\}^{nN}; y \coloneqq F(x_1, ..., x_N) \ : \ \mathcal{A}(y) \neq \perp \wedge (\forall i, x_i \in \mathsf{GOOD})\right] \\
&\quad \leq \Pr\left[(x_1, ..., x_N) \leftarrow_\$ \{0,1\}^{nN} \ : \ \forall i, x_i \in \mathsf{GOOD}\right] \\
&\quad \leq \left(1 - \frac{1}{2\mathsf{q}(n)}\right)^N = \left(1 - \frac{1}{2\mathsf{q}(n)}\right)^{2n\mathsf{q}(n)} \approx \frac{1}{e^n}
\end{aligned}$$

For the second term, fix some $j \in [N]$ and let us compute the probability that $\mathcal{A}$ inverts when $x_j \notin \mathsf{GOOD}$. Specifically, $\forall j \in [N]$:

$$\begin{aligned}
&\Pr\Big[(x_1,...,x_N) \leftarrow_{\$} \{0,1\}^{nN}; y \coloneqq F(x_1,...,x_N) \ : \ \mathcal{A}(y) \neq \perp \wedge x_j \notin \mathsf{GOOD}\Big] \\
&\quad = \Pr\Big[(x_1,...,x_N) \leftarrow_{\$} \{0,1\}^{nN}; y \coloneqq F(x_1,...,x_N) \ : \ \mathcal{A}(y) \neq \perp \ \Big| \ x_j \notin \mathsf{GOOD}\Big] \cdot \Pr[x_j \notin \mathsf{GOOD}] \\
&\quad \leq \Pr\Big[(x_1,...,x_N) \leftarrow_{\$} \{0,1\}^{nN}; y \coloneqq F(x_1,...,x_N) \ : \ \mathcal{A}(y) \neq \perp \ \Big| \ x_j \notin \mathsf{GOOD}\Big] \\
&\quad \leq N \cdot \Pr[x_j \leftarrow_{\$} \{0,1\}^n \ : \ \mathcal{B}(f(x_j)) \neq \perp \ | \, x_j \notin \mathsf{GOOD}\,] \\
&\quad \leq \frac{N}{2N\mathsf{p}(n)} = \frac{1}{2\mathsf{p}(n)}
\end{aligned}$$

By taking union bound over all j, we have a bound for the second term,

$$\begin{aligned}
&\Pr\Big[(x_1,...,x_N) \leftarrow_{\$} \{0,1\}^{nN}; y \coloneqq F(x_1,...,x_N) \ : \ \mathcal{A}(y) \neq \perp \wedge (\exists i, x_i \notin \mathsf{GOOD})\Big] \\
&\quad \leq \sum_{j=1}^{N} \Pr\Big[(x_1,...,x_N) \leftarrow_{\$} \{0,1\}^{nN}; y \coloneqq F(x_1,...,x_N) \ : \ \mathcal{A}(y) \neq \perp \wedge x_j \notin \mathsf{GOOD}\Big] \\
&\quad \leq \frac{N}{2N\mathsf{p}(n)} = \frac{1}{2\mathsf{p}(n)}
\end{aligned}$$

Now putting everything together we have,

$$\begin{aligned}
&\Pr\Big[(x_1,...,x_N) \leftarrow_{\$} \{0,1\}^{nN}; y \coloneqq F(x_1,...,x_N) \ : \ \mathcal{A}(y) \neq \perp\Big] \\
&\quad < \frac{1}{2\mathsf{p}(n)} + \frac{1}{e^n} \\
&\quad < \frac{1}{\mathsf{p}(n)}
\end{aligned}$$

This contradicts with the assumption that $\mathcal{A}$ is able to invert F with probability at least $\frac{1}{\mathsf{p}(n)}$. Thus it must be the case that the GOOD set has size at least $2^n \cdot \left(1 - \frac{1}{2\mathsf{q}(n)}\right)$.

□

This completes the proof. □

3.5 Final Remarks on OWFs

One-way functions are necessary for most of cryptography. Nevertheless, they are often not sufficient for most of cryptography. In particular, it is known that

many advanced cryptographic primitives cannot be constructed by making *black-box* use of one-way functions; however, full separations are not known.

Also, recall that we don't know if one-way functions actually exist. (We only have candidates based on assumptions such as hardness of factoring.) Now, suppose someone told you one-way functions exist (perhaps by an existence proof, and not a constructive one). Then, simply using that knowledge, could you create an explicit one-way function? Surprisingly, it *can* be done! Levin gives a proof here [Lev03].

Chapter 4

Hard Core Predicate

4.1 Introduction

In this chapter, we will first examine what information one-way functions hide, and that will introduce us to the notion of hard core predicate.

The idea of a one-way function is very intuitive, but by themselves, they're often not very useful. Why? Because it only guarantees that $f(x)$ will hide the preimage x, but no more than that! For instance, if we have a one-way function $f : \{0,1\}^n \mapsto \{0,1\}^n$, then let us consider the following function $f' : \{0,1\}^{2n} \mapsto \{0,1\}^{2n}$,

$$f'(x_1||x_2) = f(x_1)||x_2,$$

where x_1 and x_2 are of size n. It is easy to see that f' is also a one-way function, but leaks half its input!

In fact, a function f may not hide any subset of the input bits, and still be a one-way function. More generally, for any non-trivial function $a(\cdot)$, there is no guarantee that $f(x)$ will hide $a(x)$. The question that naturally follows is,

> Are there non-trivial functions of x that $f(x)$ hides?

As a starting point, we'd be happy with such a function that outputs just a single bit.

Hard Core Predicate: Intuition. A hard core predicate for a one-way function f is a function over its inputs $\{x\}$ and its output is a single bit. We want this function to be easily computed given x, but "hard" to compute given $f(x)$.

Intuitively, this says that f can leak some (or many) bits of x, but does not leak the hard core bit. In other words, finding out the hard core bit, even *given* $f(x)$, is as difficult as inverting f. Of course, we need to be a little careful with "hard to

compute" for a single bit, since it can be guessed correctly with probability $\frac{1}{2}$. So, intuitively, "hardness" should mean that it is impossible for any efficient adversary to gain any non-trivial advantage over guessing. We formalize this below:

Definition 16 (Hard Core Predicate) *Let $f : \{0,1\}^n \mapsto \{0,1\}^m$ be a one-way function. A predicate $h : \{0,1\}^* \mapsto \{0,1\}$ is a **hard core predicate** for $f(\cdot)$ if h is efficiently computable given x, and there exists a negligible function $\nu(\cdot)$ such that for every non-uniform PPT adversary $\mathcal{A}$, and $\forall n \in \mathbb{N}$,*

$$\Pr[x \leftarrow_{\$} \{0,1\}^n : \mathcal{A}(1^n, f(x)) = h(x)] \leq \frac{1}{2} + \nu(n).$$

Here the randomness is over *both* the choice of x and the random coins used by $\mathcal{A}$. As before, we also present the above definition in the form of a game.

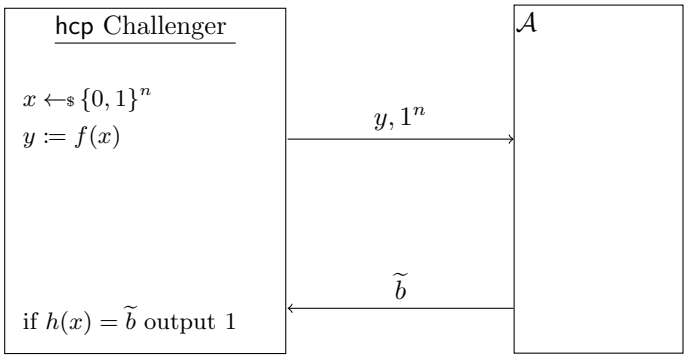


Figure 4.1: Hard-core predicate challenge game

In the above figure, we want it to hold for all non-uniform PPT $\mathcal{A}$ that hcp Challenger outputs 1 with some probability only negligible larger than $\frac{1}{2}$.

4.2 Hard Core Predicate via Inner Product

Ideally we would like every one-way function to have a hard core bit. But unfortunately, we do not know if this true. Instead, we settle for something slightly different. Namely, given *any* one-way function f, we show how to transform it into another function f' such that f' is one-way and has a hard core bit. We now describe this transformation using the inner product.

Theorem 7 (Goldreich-Levin) *Let $f : \{0,1\}^n \mapsto \{0,1\}^n$ be a one-way function (permutation). Then define $g : \{0,1\}^{2n} \mapsto \{0,1\}^{2n}$ as*

$$g(x) = f(x)||r$$

where $|x| = |r|$. Then, g is a one-way function (permutation) and

$$h(x,r) = \langle x, r \rangle$$

is a hard-core predicate for f, where $\langle x, r \rangle = (\sum_i x_i \cdot r_i) \mod 2$.

How should we prove this? If we use reduction, our main challenge is that our adversary $\mathcal{A}$ for h only outputs one bit, but our inverter $\mathcal{B}$ for f needs to output n bits. Amazingly, Goldreich and Levin proved that this can be done!

We start by considering two warmup cases, where we make assumptions on the adversary.

Assumption 2 *First, let's suppose that given $g(x,r) = f(x)||r$, adversary $\mathcal{A}$ will* always *(with probability 1) output $h(x,r)$ correctly.*

Proof We can use the property of the inner product to recover each bit of x one-by-one. For every $i \in [n]$, let e_i to be the ith standard basis vector for $\mathbb{R}^n$ (i.e., e_i is such that its ith bit is 1 but every other bit is 0). We construct an adversary $\mathcal{B}$ for f that works as follows: on input $f(x)$, It then runs the adversary $\mathcal{A}$ on input $(f(x), e_i)$ to recover x_i^*. Now, $\mathcal{B}$ simply outputs $x^* = x_1^* \cdots x_n^*$.

$\mathcal{B}(1^n, f(x))$

1 : **for** $i = 1$ **to** n do
2 : $\quad x_i^* \leftarrow \mathcal{A}(f(x), e_i)$
3 : **output** $x^* = x_1^* \cdots x_n^*$

□

Okay, that was fairly straightforward. Let's see what happens when the adversary doesn't output $h(x,r)$ correctly with probability 1.

Assumption 3 *Now assume $\mathcal{A}$ outputs $h(x,r)$ with probability $\frac{3}{4} + \varepsilon(n)$.*

Proof (Informal Strategy) The main issue here is that our adversary $\mathcal{A}$ might not work on some specific inputs - such as e_i as in the previous case. We need

the inputs to "look" random in order to mimic the expected distribution over the inputs of $\mathcal{A}$. So, we split our original single query into two queries such that each query looks random individually. Specifically, our inverter $\mathcal{B}$ computes $a := \mathcal{A}(f(x), e_i + r)$ and $b := \mathcal{A}(f(x), r)$ for $r \leftarrow\!\!\$\ \{0,1\}^n$. Then, compute $c := a \oplus b$, where $\oplus$ is `xor`. By using a union bound, one can show that $c = x_i$ with probability $\frac{1}{2} + \varepsilon$, and we can then repeatedly compute x_i and take the majority to get x_i^* with $x_i^* = x_i$ with probability $1 - \mathsf{negl}(n)$.

By repeating this process for every i, $\mathcal{B}$ can learn every x_i^* and output $x^* = x_1^* \cdots x_n^*$.

□

The full proof of Theorem 7 follows the same ideas, but needs a more careful analysis and is skipped here. Note that this theorem is actually very important, even outside cryptography! It has applications to learning, list-decoding codes, etc.

Chapter 5

Pseudorandomness

5.1 Introduction

Our computers use randomness every day, but what exactly is randomness? How does your computer get this randomness? Some common sources of randomness are key-strokes, mouse movement, and power consumption, but the amount of randomness generated by these isn't a lot, and often, especially in the context of cryptography, a lot of randomness is required. We shall see concrete instances of its usage in the subsequent chapters.

This brings us to the fundamental question: Can we "expand" a few random bits into many random bits? There are many heuristic approaches to this, but this isn't good enough for cryptography. We need bits that are *"as good as truly random bits"* (to a PPT adversary). This isn't very precise, so let's define it formally.

5.2 Computational Indistinguishability and Prediction Advantage

Suppose we have n uniformly random bits, $x = x_1 \| \cdots \| x_n$, and we want to find a deterministic polynomial time algorithm G that outputs $n+1$ bits: $y = y_1 \| \cdots \| y_{n+1}$ and looks *"as good as"* a truly random string $r = r_1 \| \cdots \| r_{n+1}$. We call such a $G : \{0,1\}^n \mapsto \{0,1\}^{n+1}$ a **pseudorandom generator**, or PRG for short.

But what does *"as good as"* really mean? Intuitively it means that there should be no obvious patterns and that it should pass all statistical tests a truly random string would pass (all possible k-length substrings should occur equally). But the key point is that we only need to address our adversary, so as long as there is no efficient test that can tell $G(x)$ and r apart, then that is enough!

This gives us the notion of **computational indistinguishability** of $\{x \leftarrow\!\!\$\ \{0,1\}^n : G(x)\}$ and $\{r \leftarrow\!\!\$\ \{0,1\}^{n+1} : r\}$. We will use this notion and an equivalent one called prediction advantage in order to define pseudorandomness. Then, we will devise a complete test for pseudorandom distributions (next-bit prediction), and examine pseudorandom generators.

First, we will have to define some terms.

Definition 17 (Ensemble) *A sequence $\{X_n\}_{n\in\mathbb{N}}$ is called an **ensemble** if for each $n \in \mathbb{N}$, X_n is a probability distribution (Definition 3) over $\{0,1\}^*$.*

Typically, we will consider X_n to be a distribution over the binary strings $\{0,1\}^{\ell(n)}$, where $\ell(\cdot)$ is a polynomial.

Now, let us try to define computational indistinguishability. This captures what it means for distributions X, Y to "look alike" to any efficient test. In other words, no non-uniform PPT "distinguisher" algorithm $\mathcal{D}$ can tell X and Y apart, or the behavior of $\mathcal{D}$ on X and Y is the same.

We can try to think of this as a game of sorts. Let's say we give $\mathcal{D}$ a sample of X. Then, $\mathcal{D}$ gains a point if it says the sample is from X, and loses a point if it says the sample is from Y. Then we can encode $\mathcal{D}$'s output with one bit. In order for X and Y to be indistinguishable, $\mathcal{D}$'s average score on a sample of X should be basically the same as its average score on a sample of Y.

$$\Pr[x \leftarrow X;\ \mathcal{D}(1^n, x) = 1] \approx \Pr[y \leftarrow Y;\ \mathcal{D}(1^n, y) = 1]$$

or

$$\left|\Pr[x \leftarrow X;\ \mathcal{D}(1^n, x) = 1] - \Pr[y \leftarrow Y;\ \mathcal{D}(1^n, y) = 1]\right| \leq \mu(n)$$

for negligible $\mu(\cdot)$.

It should be noted that in our definition, the distinguisher $\mathcal{D}$ will only get a *single* sample. This brings us to the formal definition of **computational indistinguishability**.

Definition 18 (Computational Indistinguishability) *Two ensembles of probability distributions $X = \{X_n\}_{n\in\mathbb{N}}$ and $Y = \{Y_n\}_{n\in\mathbb{N}}$ are said to be* computationally indistinguishable *if for every non-uniform PPT distinguisher $\mathcal{D}$ there exists a negligible function $\nu(\cdot)$ such that*

$$\left|\Pr[x \leftarrow X;\ \mathcal{D}(1^n, x) = 1] - \Pr[y \leftarrow Y;\ \mathcal{D}(1^n, y) = 1]\right| \leq \nu(n).$$

We can see that this formalizes the notion of a PRG if we let X be the distribution over the PRG outputs and Y be the uniform distribution over strings of the same length as PRG outputs.

But, there is actually another model for the same idea! If we give $\mathcal{D}$ a sample from either X or Y, and ask it to identify which distribution it is from, if $\mathcal{D}$ is not right with probability better than $\frac{1}{2}$, then X and Y look the same to it! Since any adversary can trivially win with probability $\frac{1}{2}$ by guessing without looking at the sample received, we want to quantify how well an adversary does beyond simply guessing. It will be convenient to change notation a bit and set $X^{(1)} = X, X^{(0)} = Y$.

Definition 19 (Prediction Advantage) *Prediction Advantage is defined as*

$$\max_{\mathcal{A}} \left| \Pr\left[b \leftarrow_{\$} \{0,1\}, t \leftarrow X_n^b : \mathcal{A}(1^n, t) = b \right] - \frac{1}{2} \right|.$$

As before, we want the prediction advantage to be negligible. Let us depict the above game pictorially as a game between a *Challenger* and an adversary $\mathcal{A}$. We say that $\mathcal{A}$ wins if the *Challenger* outputs value 1. From the prior discussion,

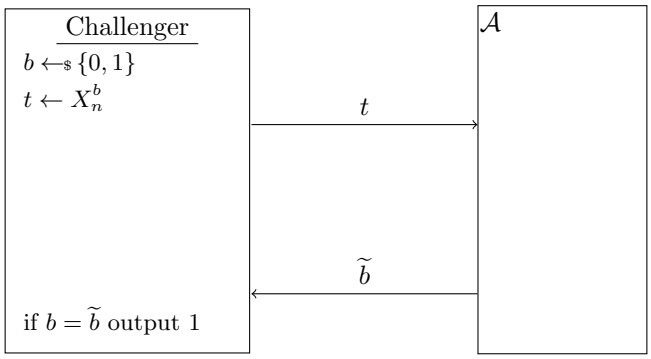


Figure 5.1: Indistinguishability Game between Challenger and adversary $\mathcal{A}$.

it is clear that probability that the challenger outputs 1 is,

$$\Pr[\text{ Challenger outputs 1}] = \Pr\left[b \xleftarrow{\$} \{0,1\}, t \leftarrow X_n^b : \mathcal{A}(t) = b \right].$$

Proposition 1 *Prediction advantage is equivalent to computational indistinguishability.*

Proof Let's write out the prediction advantage without taking the maximum over all $\mathcal{A}$,

$$\begin{aligned}
&\left|\Pr\left[b \leftarrow \{0,1\}; z \leftarrow X^{(b)}; \mathcal{D}(1^n, z) = b\right] - \frac{1}{2}\right| \\
&= \left|\Pr_{x\leftarrow X^b}\left[\mathcal{D}(x) = b \;\middle|\; b = 1\right] \cdot \Pr[b=1] + \Pr_{x\leftarrow X^b}\left[\mathcal{D}(x) = b \;\middle|\; b = 0\right] \cdot \Pr[b=0] - \frac{1}{2}\right| \\
&= \frac{1}{2}\left|\Pr_{x\leftarrow X^1}[\mathcal{D}(x)=1] + \Pr_{x\leftarrow X^0}[\mathcal{D}(x)=0] - 1\right| \\
&= \frac{1}{2}\left|\Pr[\mathcal{D}(x)=1]_{x\leftarrow X^1} + \left(1 - \Pr[\mathcal{D}(x)=0]_{x\leftarrow X^0}\right) - 1\right| \\
&= \frac{1}{2}\left|\Pr[\mathcal{D}(x)=1]_{x\leftarrow X^1} - \Pr[\mathcal{D}(x)=1]_{x\leftarrow X^0}\right|
\end{aligned}$$

So they are equivalent within a factor of 2.

In the first step of the above calculations, for any event A, we can write $\Pr[A] = \Pr[A \mid B] \cdot \Pr[B] + \Pr\left[A \;\middle|\; \overline{B}\right] \cdot \Pr\left[\overline{B}\right]$. This can in fact be extended to any partitioning of $B = B_1 \cup B_2 \cup \cdots \cup B_n$. □

A useful notion to consider for proofs is the notion of what is means for two distributions to *not* be computationally indistinguishable.

Lemma 8 (Prediction Lemma) *Let $\left\{X_n^{(0)}\right\}_{n\in\mathbb{N}}, \left\{X_n^{(1)}\right\}_{n\in\mathbb{N}}$ be ensembles of probability distributions. Let $\mathcal{D}$ be a non-uniform PPT adversary that $\varepsilon(\cdot)$-distinguishes $\{X_n^{(0)}\}, \{X_n^{(1)}\}$ for infinitely many $n \in \mathbb{N}$. Then $\exists$ a non-uniform PPT $\mathcal{A}$ such that*

$$\Pr\left[b \xleftarrow{\$} \{0,1\}, t \leftarrow X_n^b : \mathcal{A}(t) = b\right] - \frac{1}{2} \geq \frac{\varepsilon(n)}{2}$$

for infinitely many $n \in \mathbb{N}$.

Let's now look at some properties of computational indistinguishability.

Properties of Computational Indistinguishability.

1. First, we define the notation $\{X_n\} \approx_c \{Y_n\}$ to mean computational indistinguishability. Often, when clear from context we shall drop the subscript c and denote it by $\{X_n\} \approx \{Y_n\}$.
2. If we apply an efficient algorithm on X and Y, then their images under this operation are still indistinguishable. Formally, $\forall$ non-uniform PPT M,

$\{X_n\} \approx_c \{Y_n\} \implies \{M(X_n)\} \approx_c \{M(Y_n)\}$. If this were not the case, then a distinguisher could simply use M to tell $\{X_n\}$ and $\{Y_n\}$ apart!

3. If X, Y are indistinguishable with advantage at most μ_1 and Y, Z are indistinguishable with advantage at most μ_2, then X, Z are indistinguishable with advantage at most $\mu_1 + \mu_2$. This follows from the triangle inequality, and we sketch the proof below.
   We are given, for all PPT $\mathcal{D}$

$$\left| \Pr[x \leftarrow X;\ \mathcal{D}(1^n, x) = 1] - \Pr[y \leftarrow Y;\ \mathcal{D}(1^n, y) = 1] \right| \leq \mu_1(n)$$

   and

$$\left| \Pr[y \leftarrow Y;\ \mathcal{D}(1^n, y) = 1] - \Pr[z \leftarrow Z;\ \mathcal{D}(1^n, z) = 1] \right| \leq \mu_2(n).$$

   Then we compute,

$$\begin{aligned}
&\left| \Pr[x \leftarrow X;\ \mathcal{D}(1^n, x) = 1] - \Pr[z \leftarrow Z;\ \mathcal{D}(1^n, z) = 1] \right| \\
&= \Big| \Pr[x \leftarrow X;\ \mathcal{D}(1^n, x) = 1] - \Pr[y \leftarrow Y;\ \mathcal{D}(1^n, y) = 1] \\
&\quad + \Pr[y \leftarrow Y;\ \mathcal{D}(1^n, y) = 1] - \Pr[z \leftarrow Z;\ \mathcal{D}(1^n, z) = 1] \Big| \\
&\leq \left| \Pr[x \leftarrow X;\ \mathcal{D}(1^n, x) = 1] - \Pr[y \leftarrow Y;\ \mathcal{D}(1^n, y) = 1] \right| \\
&\quad + \left| \Pr[y \leftarrow Y;\ \mathcal{D}(1^n, y) = 1] - \Pr[z \leftarrow Z;\ \mathcal{D}(1^n, z) = 1] \right| \\
&= \mu_1(n) + \mu_2(n)
\end{aligned}$$

   where we use the triangle inequality that $|a + b| \leq |a| + |b|$.

This last property is actually quite nice, and we would like to generalize it a bit. The proof follows from a simple application of the pigeonhole principle and triangle inequality.

Lemma 9 (Hybrid Lemma) *Let $X^1, \ldots, X^m$ be distribution ensembles for $m = \mathrm{poly}(n)$. Suppose $\mathcal{D}$ distinguishes X^1 and X^m with advantage ε. Then $\exists i \in \{1, \cdots, m-1\}$ such that $\mathcal{D}$ distinguishes X^i, X^{i+1} with advantage $\geq \frac{\varepsilon}{m}$.*

Returning to pseudorandomness, we define a bit more notation. We call the uniform distribution over $\{0,1\}^{\ell(n)}$ by $U_{\ell(n)}$. Intuitively, a distribution is pseudorandom if it looks like a uniform distribution to any efficient test. We have the tools now to formulate this:

Definition 20 (Pseudorandom Ensembles) *An ensemble $\{X_n\}$, where X_n is a distribution over $\{0,1\}^{\ell(n)}$ is said to be pseudorandom if*

$$\{X_n\} \approx_c \{U_{\ell(n)}\}.$$

This is relevant for PRGs, as their outputs should be pseudorandom.

5.3 Next-Bit Test

In this section, we consider an interesting way to characterize pseudorandomness. For a string to be considered pseudorandom, we know that at the very least it must pass all efficient tests a true random string would pass. Let's consider one such natural test, which we shall call the "next bit unpredictability". For a truly random string, given any prefix of this string, it is not possible to predict the "next bit" with probability better than $\frac{1}{2}$. Let us extend this notion to an arbitrary string. We say a sequence of bits *passes the next-bit test* if no *efficient* adversary can predict "the next bit" in the sequence with probability better than $\frac{1}{2}$ even given all previous bits of the sequence. Let us formalize this below.

Definition 21 (Next-bit Unpredictability) *An ensemble of distributions $\{X_n\}$ over $\{0,1\}^{\ell(n)}$ is next-bit unpredictable if, $\forall\ 0 \leq i < \ell(n)$, $\forall$ non-uniform PPT adversaries $\mathcal{A}$, $\exists$ negligible function $\nu(\cdot)$ such that $\forall n \in \mathbb{N}$,*

$$\Pr\left[t = t_1 \cdots t_{\ell(n)} \leftarrow X_n : \mathcal{A}(t_1 \cdots t_i) = t_{i+1}\right] \leq \frac{1}{2} + \nu(n).$$

When presented with such a definition, an instructive exercise is to consider what it means for an ensemble to *not* be next-bit unpredictable. This will be useful when we discuss proof by reduction.

An ensemble $\{X_n\}$ is *not* next-bit unpredictable if there exists an index i, an adversary $\mathcal{A}_{\mathsf{next\text{-}bit}}$ and a polynomial $\mathsf{p}[n]$ such that for infinitely many n,

$$\Pr\left[t = t_1 \cdots t_{\ell(n)} \leftarrow X_n : \mathcal{A}_{\mathsf{next\text{-}bit}}(t_1 \cdots t_i) = t_{i+1}\right] \geq \frac{1}{2} + \frac{1}{\mathsf{p}(n)}.$$

We will refer to $\frac{1}{\mathsf{p}[n]}$ as the advantage $\mathcal{A}_{\mathsf{next\text{-}bit}}$ with respect to randomly guessing.

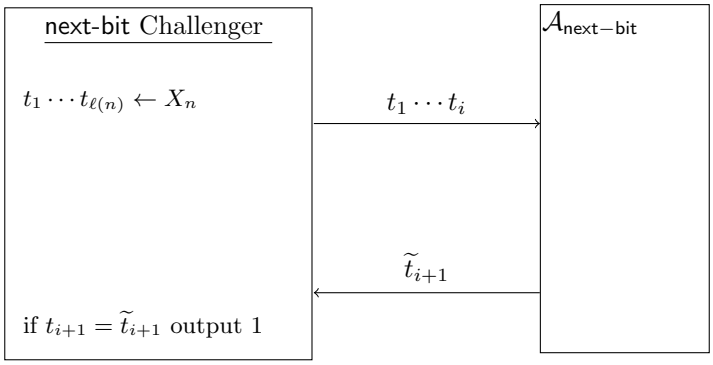


Figure 5.2: Next bit unpredictability

As before, the figure below illustrates a game for the above definition.

This intuitive test turns out to be really powerful, as we show below. As we show, the next-bit test can in fact be used to test *pseudorandomness*.

Theorem 10 (Completeness of the Next-bit Test) *If $\{X_n\}$ is next-bit unpredictable then $\{X_n\}$ is pseudorandom.*

Proof This will be our first example where we use the Hybrid Lemma (Lemma 9) in a proof. This will be an invaluable tool through this course, and therefore we shall go over this proof carefully. In subsequent proofs, we might skip some of the details with the expectation that the reader will complete them.

Let us assume to the contrary that $\{X_n\}$ is pseudorandom. From Definition 20, we have that there exists a non uniform PPT distinguisher $\mathcal{D}$, and a polynomial $p(\cdot)$ s.t. for infinitely many $n \in \mathbb{N}$, $\mathcal{D}$ distinguishes X_n and $U_{l(n)}$ with probability $\frac{1}{p(n)}$. Our goal will be to use the above distinguisher D to construct an adversary $\mathcal{A}_{\mathsf{next-bit}}$ for the next-bit unpredictability.

Let us consider the following distributions, that we shall refer to as **hybrid distributions** or **hybrids** for short. For $i \in [\ell(n)]$, we define

$$H_n^i := \left\{ x \leftarrow X_n, u \leftarrow U_{\ell(n)} : x_1 ... x_i u_{i+1} u_{i+2} ... u_{\ell(n)} \right\}$$

where $U_{\ell(n)}$ is the uniform distribution.

Note that the first hybrid H_n^0 is the uniform distribution $U_{\ell(n)}$, and the last hybrid $H_n^{\ell(n)}$ is the distribution X_n. We show the transition of the hybrids pictorially in Figure 5.3, where the darker color indicates a random bit, and the lighter a pseudorandom bit.

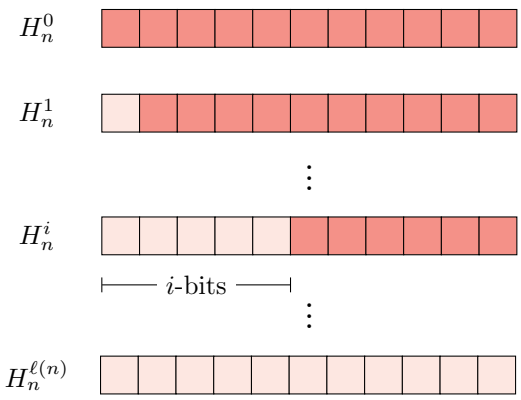


Figure 5.3: Progression of Hybrids

Think of why this is true. What would happen if it for all values of $i \in [\ell(n)]$, the probability was smaller than $\frac{1}{p(n)\ell(n)}$?

Thus, rewriting our initial claim regarding distinguisher $\mathcal{D}$ with respect to the hybrid distributions, $\mathcal{D}$ distinguishes between H_n^0 and $H_n^{\ell(n)}$ with probability $\frac{1}{p(n)}$.

By the hybrid lemma (Lemma 9), $\exists$ some $i \in [0, \ell(n)]$ s.t. $\mathcal{D}$ distinguishes between H_n^i and H_n^{i+1} with probability at least $\frac{1}{p(n)\ell(n)}$.

The only difference between H^{i+1} and H^i is that in H^{i+1}, $(i+1)^{th}$ bit is x_{i+1}, and in H^i, it is u_{i+1}.

Let's define another distribution $\tilde{H}_n^i$:

$$\tilde{H}_n^i = \left\{ x \leftarrow X_n, u \leftarrow U_{\ell(n)} : x_1 ... x_{i-1} \bar{x_i} u_{i+1} u_{i+2} ... u_{\ell(n)} \right\},$$

where $\bar{x_i} = 1 - x_i$. This is identical to H_n^{i+1} with the $i+1$-th bit flipped.

Since H_n^{i+1} can be sampled from either H_n^i or $\tilde{H}_n^i$ with equal probabilities,

$$\begin{aligned}
&\left| \Pr\left[t \leftarrow H_n^i : \mathcal{D}(t) = 1 \right] - \Pr[t \leftarrow H_n^{i+1} : \mathcal{D}(t) = 1] \right| \\
&= \Bigg| \Pr\left[t \leftarrow H_n^{i+1} : \mathcal{D}(t) = 1 \right] - \\
&\qquad \left(\frac{1}{2} \Pr\left[t \leftarrow H_n^i : \mathcal{D}(t) = 1 \right] + \frac{1}{2} \Pr\left[t \leftarrow \tilde{H}_n^i : D(t) = 1 \right] \right) \Bigg| \\
&= \frac{1}{2} \left| \Pr\left[t \leftarrow H_n^i : \mathcal{D}(t) = 1 \right] - \Pr\left[t \leftarrow \tilde{H}_n^i : \mathcal{D}(t) = 1 \right] \right|.
\end{aligned}$$

The observation that $\mathcal{D}$ distinguishes H_n^i and H_n^{i+1} with probability $\frac{1}{p(n)\ell(n)}$ implies that $\mathcal{D}$ distinguishes H_n^{i+1} and $\tilde{H}_n^{i+1}$ with probability $\frac{2}{p(n)\ell(n)}$. We shall

use $\mathcal{D}$ to construct an adversary $\mathcal{A}_{\mathsf{next-bit}}$ for the next-bit Challenger. $\mathcal{A}_{\mathsf{next-bit}}$ on receiving input $t_1 \cdots t_i$ from the next-bit Challenger, needs to prepare the input for $\mathcal{D}$. It uses $t_1 \cdots t_i$, samples a random bit b, and the rest of the string is randomly generated. If $\mathcal{D}$ responds with 1, $\mathcal{A}_{\mathsf{next-bit}}$ guesses the next bit as b, else it guesses $1 - b$. This strategy is presented in Figure 5.4. Now we need to calculate the

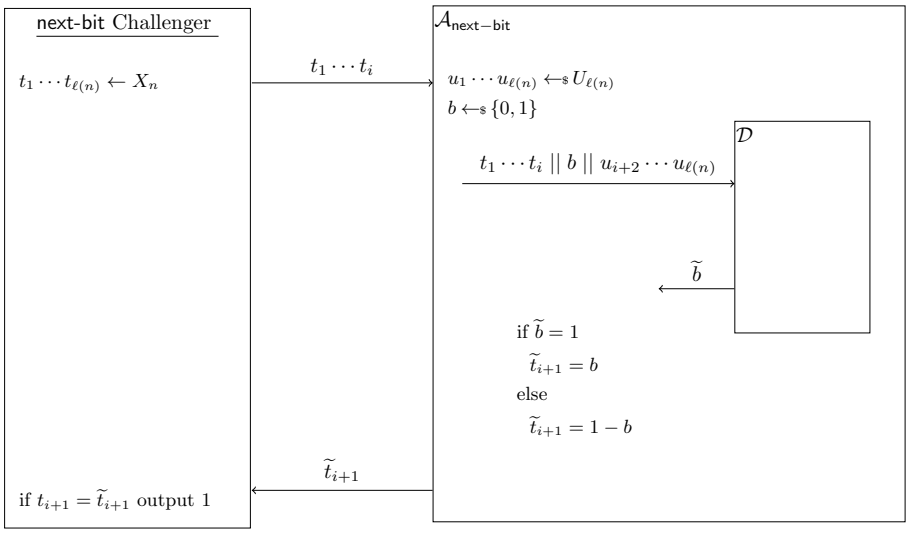


Figure 5.4: Completeness of Next-bit unpredictability

probability that $\mathcal{A}_{\mathsf{next-bit}}$ wins.

$$\begin{aligned}
\Pr[\mathcal{A}_{\mathsf{next-bit}} \text{ wins }] &= \frac{1}{2} \cdot \Pr\left[t \leftarrow H_n^i : \mathcal{D}(t) = 1\right] + \frac{1}{2} \cdot \Pr\left[t \leftarrow \widetilde{H}_n^i : \mathcal{D}(t) \neq 1\right] \\
&= \frac{1}{2} \cdot \Pr\left[t \leftarrow H_n^i : \mathcal{D}(t) = 1\right] + \frac{1}{2} \cdot \left(1 - \Pr\left[t \leftarrow \widetilde{H}_n^i : \mathcal{D}(t) = 1\right]\right) \\
&= \frac{1}{2} + \frac{1}{2} \cdot \left(\Pr\left[t \leftarrow H_n^i : \mathcal{D}(t) = 1\right] - \Pr\left[t \leftarrow \widetilde{H}_n^i : \mathcal{D}(t) = 1\right]\right) \\
&= \frac{1}{2}\frac{1}{2} \cdot \frac{1}{\mathsf{p}(n)\ell(n)}
\end{aligned}$$

This follows from the fact that with probability $\frac{1}{2}$ we will provide $\mathcal{D}$ with input either from H_n^i or $\widetilde{H}_n^i$. We have constructed an adversary $\mathcal{A}_{\mathsf{next-bit}}$ that breaks our assumption that $\{X_n\}$ is next-bit unpredictable. Therefore it must be the case that $\{X_n\}$ is pseudorandom. □

We can now rely on next bit unpredictability to prove distributions are pseudorandom.

5.4 Pseudorandom Generators (PRG)

We have defined the notion of pseudorandomness and next-bit unpredictability. Now, we turn to our goal of constructing pseudorandom generators, that will take a small amount of randomness and expand it into a large amount of pseudorandomness.

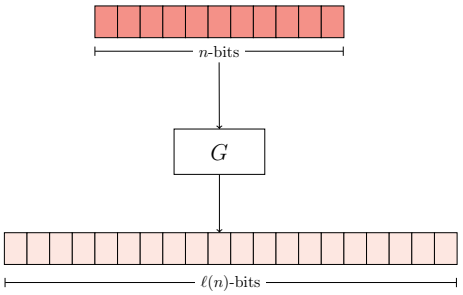


Figure 5.5: Pseudorandom Generator G

Before we can proceed with a construction, we need to define the properties we need from a pseudrandom generator.

Definition 22 (Pseudorandom Generator) *A deterministic algorithm $G : \{0,1\}^n \to \{0,1\}^{\ell(n)}$ is called a pseudorandom generator (PRG) if:*

1. *(efficiency): the output of G can be computed in polynomial time*
2. *(expansion): $|G(x)| > |x|$*
3. *(pseudorandomness): $\{x \leftarrow_\$ \{0,1\}^n : G(x)\} \approx_c \{U_{\ell(n)}\}$ where $\ell(n) = |G(0^n)|$*

A few remarks about the above definition are in order:

- **Deterministic:** It is clear that the above definition is only meaningful if G is deterministic since any additional randomness can otherwise be directly output by G.
- **Seed:** The input to the PRG is referred to as the *seed* of the PRG.
- **Randomness of the seed:** Similar to the one-way functions, the security of a PRG is defined only when the seed is chosen uniformly at random.

To evaluate the "effectiveness" of a PRG, we define the *stretch* of the PRG below.

Definition 23 (Stretch) *The stretch of G is defined as: $|G(x)| - |x|$*

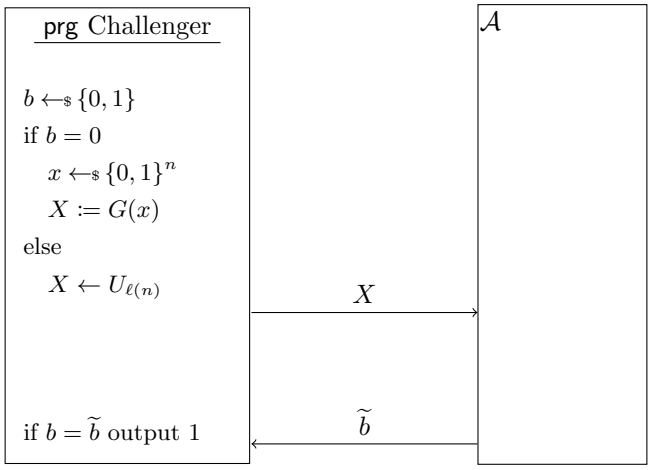


Figure 5.6: Indistinguishability Game for PRG.

We will first construct a PRG by 1-bit stretch, which already seems non trivial. Then, we will show how to generically transform a PRG with 1-bit stretch into one that achieves polynomial-bit stretch.

5.5 PRG with 1-bit Stretch

Armed with what we've learned so far, a natural candidate for our construction of PRG is

$$G(s) = f(s)||h(s)$$

where f is a one-way function and h is the hardcore predicate associated with f.

While $h(s)$ is indeed unpredictable even given $f(s)$, this construction is not really a PRG because of the following reasons:

- $|f(s)|$ might be less than $|s|$.
- $f(x)$ may always start with a prefix, which is not random. Indeed, OWF doesn't promise random outputs.

It turns out that PRGs can in fact be constructed from one-way functions, but the construction is quite involved. Instead, we'll settle on a slightly easier problem of constructing PRG from a one-way *permutation* (OWP) over $\{0,1\}^n$.

A one-way permutation clearly address both of the above issues:

- Since f is a permutation, the domain and range have the same number of bits, i.e., $|f(s)| = |s| = n$.

– $f(s)$ is uniformly random (not just pseudorandom) over $\{0,1\}^n$ if s is randomly chosen. In particular $\forall y$:

$$\Pr[s \leftarrow\!\!\$\, \{0,1\}^n : f(s) = y] = \Pr\left[s \leftarrow\!\!\$\, \{0,1\}^n : s = f^{-1}(y)\right] = \frac{1}{2^n}.$$

Thus, if s is uniform, $f(s)$ is uniform.

As it turns out, our initial proposed construction for works when f is a one-way permutation. Let us now prove the following theorem:

Theorem 11 (PRG based on OWP) *Let f be a one-way permutation, and h be a hard-core predicate for f. Then $G(s) = f(s)||h(s)$ is a PRG with 1-bit stretch.*

Proof Let us assume, to the contrary, that G is not a PRG. Then from Definition 22, we know that the output of G is not pseudorandom. Since we've established that the next-bit unpredictability is complete (Theorem 10), if the output of G is not pseudorandom, it must not satisfy next-bit unpredictability.

Putting this all together, if G is not a PRG, there exists an index i, a non-uniform PPT adversary $\mathcal{A}_{\mathsf{next-bit}}$, a polynomial $\mathsf{p}(\cdot)$ such that for infinitely many values of n,

$$\Pr[s \leftarrow\!\!\$\, \{0,1\}^n, y_1 \cdots y_n y_{n+1} \coloneqq G(s) : \mathcal{A}_{\mathsf{next-bit}}(y_1 \cdots y_i) = y_{i+1}] = \frac{1}{2} + \frac{1}{\mathsf{p}(\cdot)}.$$

Let us take a closer look at what values i can take. From our construction, we know that if s is random, the first n bits are random and no adversary can guess any of its bits with probability better than $\frac{1}{2}$. So it must be the case that $i = n$. We will use this information to construct an adversary $\mathcal{A}_{\mathsf{hcp}}$ for the hard-core predicate. The strategy is quite simple, when $\mathcal{A}_{\mathsf{hcp}}$ is given $y \coloneqq f(x)$, this is passed along to $\mathcal{A}_{\mathsf{next-bit}}$ and the corresponding response from $\mathcal{A}_{\mathsf{next-bit}}$ is used as the response to the hcp Challenger. We describe this pictorially below:

From the description above it is clear that $\mathcal{A}_{\mathsf{hcp}}$ succeeds if and only if $\mathcal{A}_{\mathsf{next-bit}}$ succeeds. Therefore, the probability $\mathcal{A}_{\mathsf{hcp}}$ succeeds is given by

$$\Pr[\mathcal{A}_{\mathsf{hcp}} \text{ wins }] = \Pr[\mathcal{A}_{\mathsf{next-bit}} \text{ wins }] \geq \frac{1}{2} + \frac{1}{\mathsf{p}(n)}.$$

This contradicts the assumption that h is a hard-core predicate for f. Thus, it must be the case that G is a PRG. □

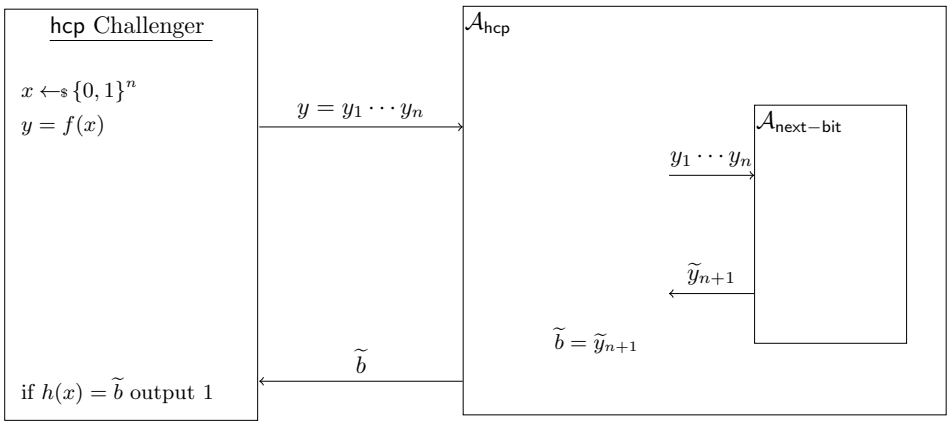


Figure 5.7: 1-bit Stretch PRG

5.6 PRG with Poly-Stretch

To be useful, we will need to go beyond a 1-bit stretch, to any arbitrary polynomial. From our previous section, why stop at 1 bit? Why don't we repeatedly apply the same procedure. Indeed, this will be how we will construct our PRG. See Figure 5.8 for visual representation of this idea, where we collect the last bit from each iteration and this forms the output of the PRG G. The first n bits from each output are fed into the next iteration of G.

We formally prove the lemma below.

Lemma 12 *Let $G : \{0,1\}^n \to \{0,1\}^{n+1}$ be a PRG. For any polynomial ℓ, $G' : \{0,1\}^n \to \{0,1\}^{\ell(n)}$ is defined as:*

$$G'(s) = b_1 ... b_{\ell(n)} \text{ where}$$
$$X_0 := s$$
$$X_{i+1} || b_{i+1} := G(X_i)$$

Then, G' is a PRG.

Proof We first establish some notation. Let $G'(s) = G^{\ell(n)}(s)$, where

$$G^0(x) = \epsilon$$
$$G^i(x) = b || G^{i-1}(x') \text{ where } x' || b := G(x)$$

and ϵ denotes the empty string.

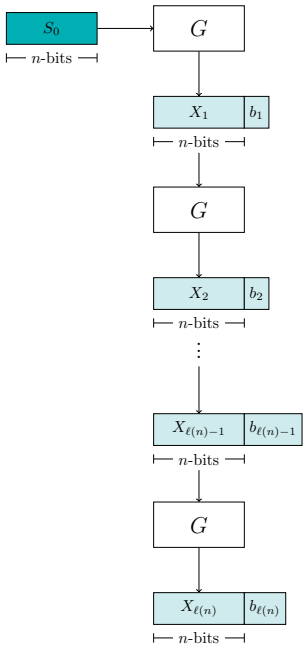


Figure 5.8: PRG with polynomial Stretch from PRG with a 1-bit Stretch.

Let us assume, to the contrary, that G' is not a PRG. Then, from Definition 22, there exists a distinguisher $\mathcal{D}$, and a polynomial $\mathsf{p}(\cdot)$ such that for infinitely many n, $\mathcal{D}$ distinguishes

$$\left\{U_{\ell(n)}\right\} \text{ and } \{G'(U_n)\}$$

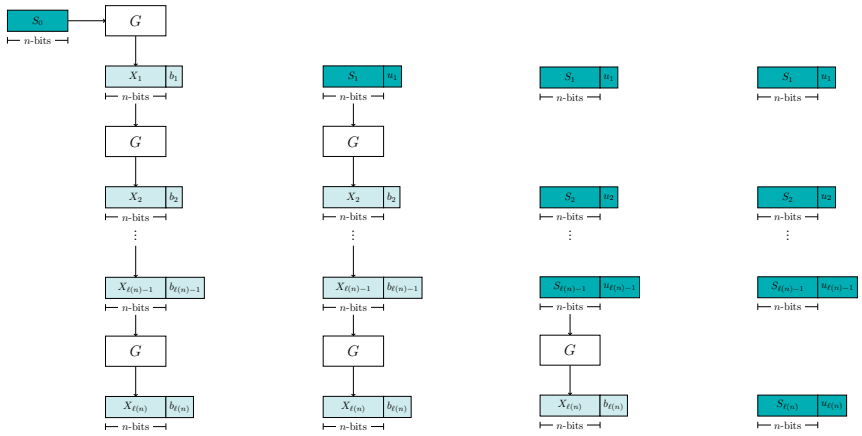
with non-negligible probability $\frac{1}{p(n)}$.

We shall use the Hybrid Lemma (Lemma 9) as the main tool for our proof. To do so, let us define the following hybrid distributions $\forall i \in [\ell(n)]$:

$$H_n^i = U_i \| G^{\ell(n)-1}(U_n),$$

i.e. in hybrid H_n^i, the first i bits are random, while the rest are obtained as in the process above. Figure 5.9 gives a visual depiction of these hybrids. Then, $H_n^0 = G^{\ell(n)}(U_n)$ and $H_n^{\ell(n)} = U_{\ell(n)}$ and $\mathcal{D}$ distinguishes H_n^0 and $H_n^{\ell(n)}$ with probability $\frac{1}{\mathsf{p}(n)}$.

By the hybrid lemma, for infinitely many n, there exists index i such that $\mathcal{D}$ distinguishes H_n^i and H_n^{i+1} with probability $\frac{1}{\ell(n)\mathsf{p}(n)}$. Since $\ell(n)$ is polynomial, so is $\ell(n)\mathsf{p}(n)$



(a) H_0 (b) H_1 (c) $H_{\ell(n)-1}$ (d) $H_{\ell(n)}$

Figure 5.9: Hybrids in

Note that the only difference between H_n^i and H_n^{i+1} can be seen by writing out the hybrids explicitly

$$\begin{aligned} H_n^i &= U_i \| G^{\ell(n)-i}(U_n) \\ &= U_i \| b \| G^{\ell(n)-i-1}(x) \\ H_n^{i+1} &= U_{i+1} \| G^{\ell(n)-i-1}(U_n) \\ &= U_i \| U_1 \| G^{\ell(n)-i-1}(U_n) \end{aligned}$$

where $x \| b \coloneqq G(U_n)$. Thus, only the $i+1$-th bit either comes from a PRG output or is truly random. Now we shall use the adversary $\mathcal{D}$ to build an adversary $\mathcal{A}_{\mathsf{prg}}$ that breaks the pseudorandomness of G. $\mathcal{A}_{\mathsf{prg}}$ gets as input either a random $n+1$ bit string, or an output of a PRG of the same length. The strategy is to provide to $\mathcal{D}$ as input something from either H_n^i or H_n^{i+1} using the inputs $\mathcal{A}_{\mathsf{prg}}$ receives. From the construction, and the description of the hybrids, it is clear that we should use the input to $\mathcal{A}_{\mathsf{prg}}$ for the $i+1$-th bit for the input to $\mathcal{D}$. Formally, this is shown in the diagram below, If $b = 0$, then the input constructed by $\mathcal{A}_{\mathsf{prg}}$ for $\mathcal{D}$ is identical to H_n^i, while if $b = 1$, then it is identical to H_n^{i+1}. Therefore, the value $\widetilde{b}$ returned by $\mathcal{D}$ also indicates whether the input to $\mathcal{A}_{\mathsf{prg}}$ was output of a PRG or random. In fact $\mathcal{A}_{\mathsf{prg}}$ has the same probability of success as $\mathcal{D}$. Therefore,

$$\Pr[\mathcal{A}_{\mathsf{prg}} \text{ wins }] = \Pr[\mathcal{D} \text{ wins }] \geq \frac{1}{2} + \frac{1}{\ell(n)\mathsf{p}(n)}.$$

Since G runs in polynomial time, so does $\mathcal{A}_{\mathsf{prg}}$. Therefore our constructed $\mathcal{A}_{\mathsf{prg}}$

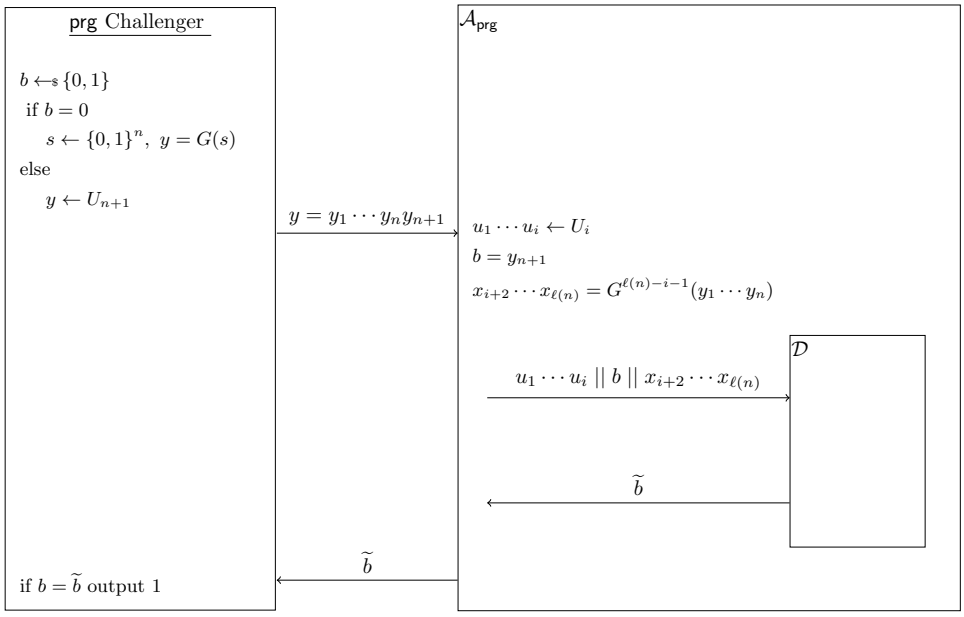


Figure 5.10: 1-bit to poly-bit stretch PRG

What happened if we got greedy and tried to output say X_1? Would it still be secure?

contradicts the assumption that G is a PRG. It must then be the case that G' is a PRG.

□

5.7 Going beyond Poly Stretch

PRGs can only generate polynomially long pseudorandom strings. What if we want exponentially long pseudorandom strings? How can we efficiently generate them? One way to do this is by using functions that index exponentially long pseudorandom strings.

Towards that end, let us start by defining a random function? Consider a function $f : \{0,1\}^n \to \{0,1\}^n$. If we write f as a table, where first column has input strings from 0^n to 1^n and the second column has the function value against each input, each row of the table is of the form $(x, f(x))$. As we've seen in Example 2, the total number of functions that map n bits to n bits is 2^{n2^n}.

To define a random function, we can use one of the two methods:

1. Select a random function F *uniformly at random* from all 2^{n2^n} functions that map n bits to n bits

2. Use a randomized algorithm to describe the function. This is called *lazy sampling*, and often more convenient implementation for proofs. Specifically,
   - A randomized program M keeps a table $\mathcal{T}$ (initially empty) to implement a random function F
   - On input x,
     - if (x, y') is not in the table, choose a random string y and add the entry (x, y) to $\mathcal{T}$
     - otherwise, M outputs y'.

Note that the distribution of M's output is identical to that of a random function F.

However, truly random functions are huge random objects. Neither of the methods allows us to store the entire function efficiently. But with the second method, M will only need polynomial space and time to store and query $\mathcal{T}$, if one makes **polynomial** calls to the random function.

Pseudorandom Functions (PRF): Intuition. PRF looks like a random function and is described in polynomial bits. At first, it seems like a good idea to use computational indistinguishability to make PRF "look like" a random function. However, the main issue with this idea is that a random function is of an exponential size. If a distinguisher $\mathcal{D}$ can't even read the input efficiently in one case, then it can distinguish between PRFs and random functions by looking at its input size, and computational indistinguishability is violated. One way to solve this issue is to allow $\mathcal{D}$ to only *query* the function on inputs of its choice, and let it see the output.

5.8 Pseudorandom Functions (PRF)

PRFs look like random functions, but need only polynomial number of bits to be described and emulated. What does "look like" really mean? Let us use computational indistinguishability to define it. But the question that arises here is that, can the distinguisher $\mathcal{D}$ be given the entire description of a random function or a PRF?

Issue: Since the description of a random function is of exponential size, $\mathcal{D}$ might not even be able to read the input efficiently in case of random function and can easy tell the difference just be looking at the size of input.

Suggested Solution: What if $\mathcal{D}$ is not given the description of functions as input, but is instead allowed to only query the functions on inputs of its choice, and view the output. But the question here is whether the entire implementation of the PRF is hidden from the distinguisher or only a part of it.

Keeping the entire description of PRF secret from $\mathcal{D}$, is similar to providing security by obscurity which in general is not a good idea according to Kerchoff's principle.

Therefore we use the notion of keyed functions. It is better to define PRFs as keyed functions. So only the key remains secret while the PRF evaluation algorithm can be made public. This is in accordance with Kerchoff's principle. The security of PRF is defined via game base definition.

5.8.1 Security of PRF via Game Based Definition

There are two players, a PRF Challenger and an Adversary/Distinguisher $\mathcal{D}$.

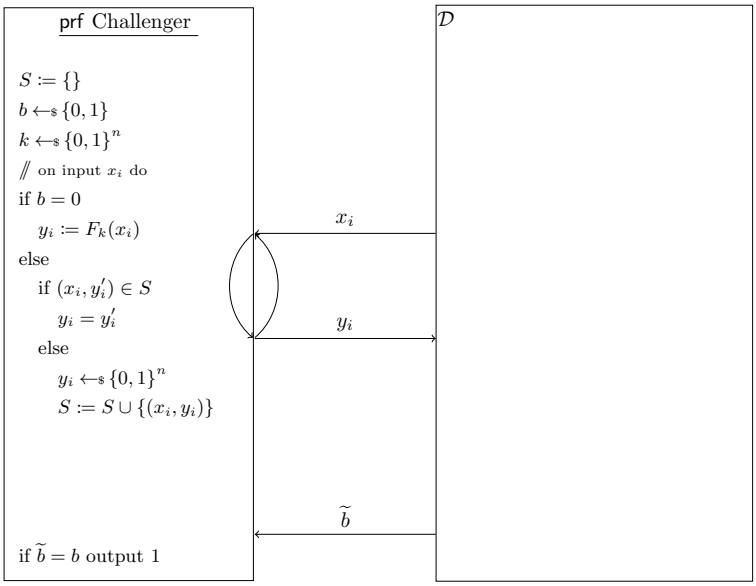


Figure 5.11: PRF Challenger Game

If $b = \widetilde{b}$, $\mathcal{D}$ wins the game. Intuitively, we want that the adversary wins with probability only negligibly greater than $\frac{1}{2}$.

Definition 24 (Pseudorandom Functions) *A family $\{F_k\}_{k\in\{0,1\}^n}$ of functions, where $F_k : \{0,1\}^n \to \{0,1\}^n$ for all k is pseudorandom if, it is:*

- ***Easy to Compute:*** *There is an efficient algorithm M such that $\forall k, x$: $M(k,x) = F_k(x)$.*
- ***Hard to Distinguish:*** *For every non-uniform PPT $\mathcal{D}$, there exists a negligible function ν such that $\forall n \in N$:*

$$Pr[\mathcal{D} \text{ wins}] \leq \frac{1}{2} + \nu(n) \tag{5.1}$$

We say that $\mathcal{D}$ wins the game if the prg Challenger outputs 1.

Remark 6 *We are ensuring that the challenger is efficient by implementing the random function using the second method. It is important to construct challengers that are efficient, because while building reductions, the reduction acts as the challenger. If the challenger is not efficient, it would lead to the construction of an inefficient reduction. Since we only consider n.u. PPT adversaries, an inefficient reduction would not be able to give us a contradiction.*

5.8.2 PRF with 1-bit input

Intuitively, PRFs with 1- bit input, can be constructed using PRGs. Since PRFs are keyed functions, the key of PRF can be used as the random seed for PRG. We can construct a PRF $F_k : \{0,1\} \to \{0,1\}^n$ using a length doubling PRG, G as follows:

- Let G be a length doubling PRG such that, $G(s) = y_0||y_1$, where $|y_0| = |y_1| = n$.
- For PRF, set $s = k$, and $F_k(0) = y_0$, $F_k(1) = y_1$

The security of PRFs constructed like this, can be argued by the security of PRGs.

Proof Since this PRF is directly using the output of PRG, we can say that if there exists a distinguisher A that can efficiently distinguish between a random F and this PRF, then another distinguisher B can be constructed using A, that will be able to distinguish between PRG and a random number generator in the following way:

1. B gets a 2n- bit input y which is either a random string or the output of G for some s.

2. The adversary A, which is a distinguisher for the PRF, is allowed to query on inputs of its choice. So if A sends a query for input "0", B sends the first n-bits of y to A. On input query "1", B sends the last n-bits of y to A.
3. Based on the queries and their responses, A either outputs "1" (if the function is a PRF) or "0" (if the function is random).
4. B outputs "0" if the output of A was "0", and it outputs "1", if the output of B was "1".

Since the distinguisher A can distinguish between a random F and PRF with noticeable probability. From the above construction, it follows that B can also distinguish between a random string and the output of a PRG with noticeable probability. But, we know that no such distinguisher for PRGs exists, therefore, no such distinguisher for this PRF can exist. □

5.8.3 PRF with n-bit input

Total number of possible n-bit inputs are exponential. Since PRGs only stretch to poly(n) bits, a single PRG with poly stretch can only be used for constructing PRFs with $log(n)$ bit input. F_k: $\{0,1\}^{log(n)} \rightarrow \{0,1\}^n$ can be constructed using $G(s) = y_1||y_2||...||y_{L=poly(n)}$, where $|y_i|$ = n-bits.
However, For n-bit input, the "double and choose" policy used for constructing PRFs with 1-bit input can be used multiple times, repeatedly.

Theorem 13 (Goldreich-Goldwasser-Micali (GGM)) *If pseudorandom generators exist, then pseudorandom functions exist.*

Construction:
For a length doubling PRG G, we define G_0 and G_1 as: $G_0(s) = G(s)[1:n]$, and $G_1(s) = G(s)[n+1:2n]$.
For n-bit input $x = x_1x_2...x_n$,

$$F_k(x) = G_{x_n}(G_{x_{n-1}}(.....(G_{x_1}(k))..)$$

It is convenient to think of the construction of F_k as a binary tree of size 2^n.
k denotes the root (chosen randomly). While $k_{x_1....x_l}$, denotes the the leaves of the tree at level l.
At level l, there are 2^l nodes, one for each path, denoted by $k_{x_1....x_l}$. The evaluation of function F_k on an input string $x_1x_2....x_n$ can be thought of as a walk down the branches of the tree up till the leaf nodes. Starting from the root, based on x_0, either the path traverses through the left branch or the right branch. Subsequently, the choice of branches is based on individual bits of the input string. Every input

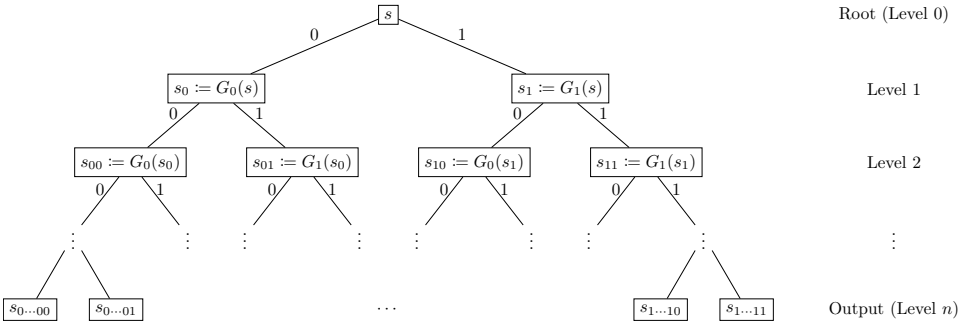


Figure 5.12: PRF Construction

would correspond to a unique path and a unique leaf node, since atleast one bit (and correspondingly, one edge in the tree) would be different.

Efficiency: Considering the efficiency of this construction, there is no need to store this tree, the output values can be computed on the fly. To compute an output value,
RunningTime($F_k(x)$) = $n\times$ RunningTime($G(.)$) = $n \times poly(n)$ = $poly'(n)$.

Proof Strategy:
The natural intuition is to prove using Hybrid arguments. The first idea to construct hybrids, is to replace each node in the tree, from the output of a PRG to a random string one by one. But since there are exponential number of nodes in the tree, this would result in exponential number of hybrids. While we know that the Hybrid Lemma, only works for polynomial number of hybrids. To overcome this, we use an interesting observation, which is that any PPT adversary is only allowed to make polynomial queries. Since each query corresponds to a unique path (of n nodes), total number of nodes affected by all queries = $n \times poly(n) = poly(n)$. Therefore, we only need to change polynomial number of nodes in the tree. We can achieve this by using 2 layers of nested hybrids.

Proof Level 1 Hybrids:
H_0 : Level 0 is random, level $i > 0$ is pseudorandom. (actual PRF)
H_1 : Level 0,1 are random, level $i > 1$ is pseudorandom.
..
..
H_i : Levels $\leq i$ are random, levels $> i$ are pseudorandom.
..

..
H_n : all levels are random. (random function)

Remark. By saying that level i is random or pseudorandom, we mean that all the nodes at level i, that are affected by the adversary's queries are random or pseudorandom respectively. H_0 corresponds to a PRF, while H_n corresponds to a random function. if we assume that PRF and random function are distinguishable, then $\exists D$, such that it can differentiate between H_0 and H_n with noticable advantage $\epsilon(n)$, then by Hybrid Lemma, $\exists i \in [n]$, such that $\exists D'_{H_i,H_{i+1}}$ that distinguishes between H_i and H_{i+1} with advantage atleast $\frac{\epsilon(n)}{n}$. The only difference between H_i and H_{i+1} is that:
- in H_i, level $i+1$ is pseudorandom.
- in H_{i+1}, level $i+1$ is random.
We need to create another set of hybrids between H_i and H_{i+1}.

To create these hybrids, we only need to replace the nodes that are affected by the queries of the adversary. Since the number of queries are polynomial, changing polynomial number of nodes is sufficient from adversary's point of view. Let S be the set of nodes at level i that are affected by the adversary/distinguisher's input queries, i.e., $|S| = poly(n)$.

Level 2 Hybrids (assuming all nodes in S are in lexicographic order):
$H_{i,j\in|S|}$: Same as H_i, except that all nodes at level $i+1$, that are children of nodes $\leq j$, are random.
$H_{i,0} \equiv H_i$ (a dummy hybrid)
$H_{i,|S|} \equiv H_{i+1}$

Given $D'_{H_i,H_{i+1}}$, by Hybrid Lemma, $\exists j \in |S|$ and $\exists D''_{H_{i,j},H_{i,j+1}}$ such that, it distinguishes between $H_{i,j}$ and $H_{i,j+1}$ with advantage atleast $\frac{\epsilon(n)}{n\times|S|} \geq \frac{1}{poly(n)}$ (recall that $\epsilon(n)$ is noticeable (i.e., $\geq \frac{1}{poly(n)}$)). The only difference between $H_{i,j}$ and $H_{i,j+1}$ is that:
- in $H_{i,j}$, node j in level $i+1$ is pseudorandom.
- in $H_{i,j+1}$, node j in level $i+1$ is random.
If $D''_{H_{i,j},H_{i,j+1}}$ can distinguish between $H_{i,j}$ and $H_{i,j+1}$ with a noticeable advantage, then it is possible to construct another adversary D''' that can efficiently distinguish between PRG and a random number generator as follows:

1. D''' gets a 2n-bit input $y_0||y1$ ($|y_0| = |y_1|$), that is sampled either as a random string or as $G(s)$ for some random string s.

2. In his mind, D''' chooses random n-bit strings for nodes that are affected by the adversary's queries, up till level i and for nodes at level $i+1$ that are children of nodes $< j$. The children of node j are substituted by y_0 and y_1 respectively. The remaining nodes in the tree that are affected by the adversary's queries as computed using the chosen random values and y_0 and y_1. D''' replies to the queries of $D''_{H_{i,j},H_{i,j+1}}$ using this tree.
3. If y_0 and y_1 were pseudorandom the distribution of D''''s outputs would be similar to $H_{i,j}$ and they were random then, the output distribution would be similar to $H_{i,j+1}$.
4. If $D''_{H_{i,j},H_{i,j+1}}$ decides that the distribution of these nodes matches the distribution of $H_{i,j}$, then D''' decides that the input is pseudorandom, else if it matches the distribution of $H_{i,j+1}$, then D''' decides that the input is random.

According to the above construction, B can efficiently distinguish between a random string and the output of a PRG. Since we know that no such distinguisher for PRGs exists, therefore no $D''_{H_{i,j},H_{i,j+1}}$ can exist. From earlier definitions and assumptions, it follows that no distinguisher $D'_{H_i,H_{i+1}}$ for $H_i(H_{i,0})$ and $H_{i+1}(H_{i,|S|})$ can exist. And subsequently no D that distinguishes between a random function and a PRF can exist. Which implies that this construction of PRF is indistinguishable. □

Bibliography

[Bar] Boaz Barak. http://www.boazbarak.org/cs127/chap001-mathematical-background.pdf.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[Kat] Jonathan Katz. http://www.cs.umd.edu/~jkatz/complexity/f11/all.pdf.

[Lev03] Leonid A. Levin. The tale of one-way functions. *Probl. Inf. Transm.*, 39(1):92–103, 2003.

[Lov] Andrew D Loveless. https://sites.math.washington.edu/~aloveles/Math300Summer2011/m300Quantifiers.pdf.