CSE 594 : Modern Cryptography 2/23/2017

Pseudorandomness-III

Instructor: Omkant Pandey *Scribes: Gustavo Poscidonio, Justin Maldonado*

1 Random Functions

We begin by considering how to define a random function. These functions F will be in the form $F : \{0,1\}^n \to \{0,1\}^n$. It will be useful for future constructions to think of F as a table:

x	$F(x)$
$000\ldots000$	$101\ldots110$
$000\ldots001$	$111\ldots010$
$000\ldots010$	$001\ldots110$
$\vdots$	$\vdots$
$111\ldots101$	$011\ldots000$
$111\ldots110$	$101\ldots001$
$111\ldots111$	$001\ldots011$

For each input x we select a random sequence of bits and we call that sequence $F(x)$. This table is exponentially large with 2^n entries in the table. Each entry in the image of F (i.e. the right column of the table) takes up n bits. Thus, a bitwise representation of F would take up $n \cdot 2^n$ bits. Note that there are a total of $2^{\text{size of } F} = 2^{n2^n}$ such functions F that map n bits to n bits.

One way to define a random function is to select *uniformly at random* one of these 2^{n2^n} functions. This is impractical because just describing such a function (e.g., as a table of entries) takes exponential time. Since the function is truly random, its full table-description cannot be compressed too much. Another alternative to define the function is as follows:

Describing F in Polynomial Time Use a randomized algorithm M to iteratively describe the function:

1. M initializes a function table (as pictured above) called T. This table T is initially empty.

x	$F(x)$

2. On input a to M, is M does not have an entry for a in its table, choose a random string b and add the entry (a, b) to the table.

x	$F(x)$
a	b

3. If a is already in the table, then simply query the table T for the image of a. So in the table above, $F(a) = b$.

Observe here that M's output distribution is identical to that of F, i.e. it is truly random.

The inherent problem with truly random functions is that they are incredibly large objects. Consider storing an entire truly random function that maps from 4 bits to 4 bits. That requires $2^{4 \cdot 2^4}$ bits $\approx$ 2 million terabytes. It doesn't matter how we construct T, we can't possibly store the entire function efficiently. The method we described to generate T one entry at a time only allows us to support **polynomial** calls to the function. It doesn't magically let us define the whole function in polynomial space or time.

2 Pseudorandom Functions (PRF)

A PRF is a construct which looks like a random function but only requires polynomially many bits to be described. Precisely speaking, a PRF and a random function should be computationally indistinguishable.

So let us consider a distinguisher D that will try to distinguish between a PRF and a random function. If we simply give D the full description of the PRF and the random function:

1. D won't be able to read the input efficiently
2. D can distinguish between the PRF and the random function by simply looking at the size of the inputs. The larger one will be the random function with high probability.

The solution to these issues is that D should only be able to *query* the function on inputs of its choice and see the output. Now, that's not to say that the description of the PRF should be hidden from D. This violates Kerckoff's principle which says that security by obscurity is a bad idea. So instead we will make the PRF a keyed function. This is far more ideal since if the key is exposed, it's easy to simply generate a new key. If an algorithm gets exposed, it's not easy to generate a new algorithm. So we use a publicly understood algorithm with a secret key as our PRF.

We will define the security of a PRF using a **Game based definition**. The Game will have two players: a **challenger** Ch and a distinguisher D.

1. The Game begins with Ch choosing a random bit b. If $b = 0$, then Ch will implement a random function. Otherwise it will implement a PRF. This selection remains for the duration of the Game. Ch may not switch the function it is using once the Game has started.
2. D sends queries $x_1, x_2, \ldots$ to Ch one at a time.
3. Ch correctly applies whichever function it chose and responds with the result $F(x_1), F(x_2), \ldots$.
    - Observe that PRFs run in polynomial time. Also observe that the calculation $F(x)$ where F is a random function can also be done in polynomial time if we use the iterative construction of F as given earlier in the lecture. So regardless of whether Ch implements a PRF or a random function, we can compute $F(x)$ in polynomial time.
4. After however many queries D chooses to make, D will output it's guess b'. That is, D guesses whether the output came from a PRF or a random function.

5. D wins of $b' = b$.

We say that PRFs are secure if no D can win with probability greater than $\frac{1}{2}$

Definition 1 *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:*

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

$$\left|\Pr[D \text{ wins } \mathsf{GuessGame}\,] - \frac{1}{2}\right| \leq \nu(n)$$

where GuessGame *is defined below:*

GuessGame(1^n) incorporates D and proceeds as follows:

- The games choose a PRF key k and a random bit b.
- It runs D answering every query x as follows:
- If $b = 0$: (answer using PRF)
  - output $F_k(x)$
- If $b = 1$: (answer using a random F)
  - (keep a table T for previous answers)
  - if x is in T: return $T[x]$
  - else: choose $y \leftarrow \{0,1\}^n$, $T[x] = y$, return y
- Game stops when D halts. D outputs a bit b'.

D **wins the** GuessGame **if** $b' = b$.

3 Constructing a PRF

We will be constructing a PRF using a PRG. We can generalize this later so we will begin by building a PRF for just 1-bit inputs using a PRG. Our construction begins as follows:

Let G be a length doubling PRG. Our goal is to find some family of functions $\{F_k\}$ over all keys k such that $F_k : \{0,1\} \to \{0,1\}^n$. We know that G is length doubling so let us write:

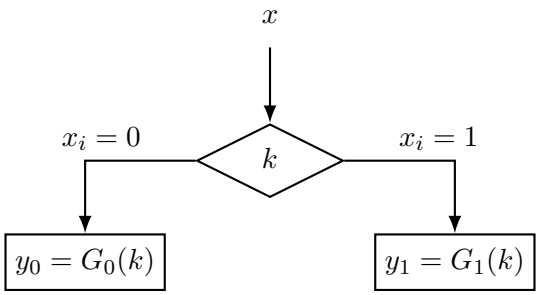
$$G(s) = y_0||y_1$$

where $|y_0| = |y_1| = n$. The idea here is that on 1-bit input x to some function F_k, if $x = 0$ then we output y_0, otherwise we output y_1. More precisely, let $b \leftarrow \{0,1\}$, then:

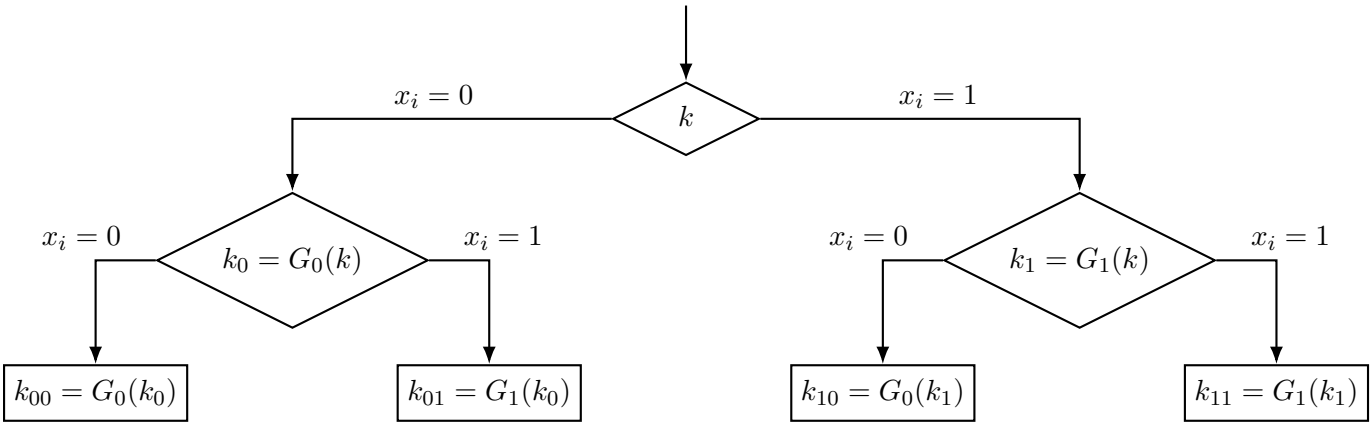
$$F_k(b) = y_b \text{ where } G(k) = y_0||y_1$$

By constructing F in this way, we guarantee that for any 1-bit input x, $F_k(x)$ will always be the same, and it will always look random since G always produces random looking output.

We can extend the ideas discussed here to work for n-bit inputs. For the 1-bit case, we applied the idea of "double and choose" where we selected the left side of G's output if the input was 0 and the right side if the input was 1. We may visualize this as a decision tree with depth 1 (Note: in the following diagrams, i represents the depth of the tree at that node):



We may expand this to an input x with n-bits by simply expanding the decision tree. Consider an input x of length 2. The decision tree would be as follows:



The idea here is that each x will take a unique path down this tree. That is, you will repeatedly "double and choose" and your choice is dependent on the ith bit of x. Eventually, you will end at a leaf node with the label k_x. For example, if $x = 01$, then the PRF described by the decision tree above would output $F_k(01) = k_{01}$.

Theorem 1 *If pseudorandom generators exist, then pseudorandom functions exist.*

Before beginning our proof, let us clarify some notation about our construction. We define G_0 and G_1 as:

$$G(s) = G_0(s)||G_1(s)$$

i.e., G_0 chooses the left half of G and G_1 chooses the right half. Now for $x \in \{0,1\}^n$ we write $x = x_1x_2 \dots x_n$ where x_i is the ith bit of x. Then we define $F_k(x)$ as:

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

Observe that this is merely saying that for each $i \leq n$, we let the ith bit of x determine whether we choose the left half of G or the right half of G. Consult the tree above for the decision tree generated by this function where $n = 2$.

Proof.

We will pursue a proof by hybrid arguments. Observe, however, that if we try to create hybrids on each possible leaf node in the tree, then we will have exponentially many hybrids and the Hybrid Lemma won't hold. So we must coneive of a way to construct polynomially many hybrids. Here, we make the observation that an efficient adversary can only make polynomially many queries. Thus, we really only need to change polynomially many nodes in the tree. So we will construct a hybrid over an arbitrary path taken by some input x.

Suppose to the contrary that $F_k(x)$ is not pseudorandom. Formally, this means that for some distinguisher D and some noticeable function ε:

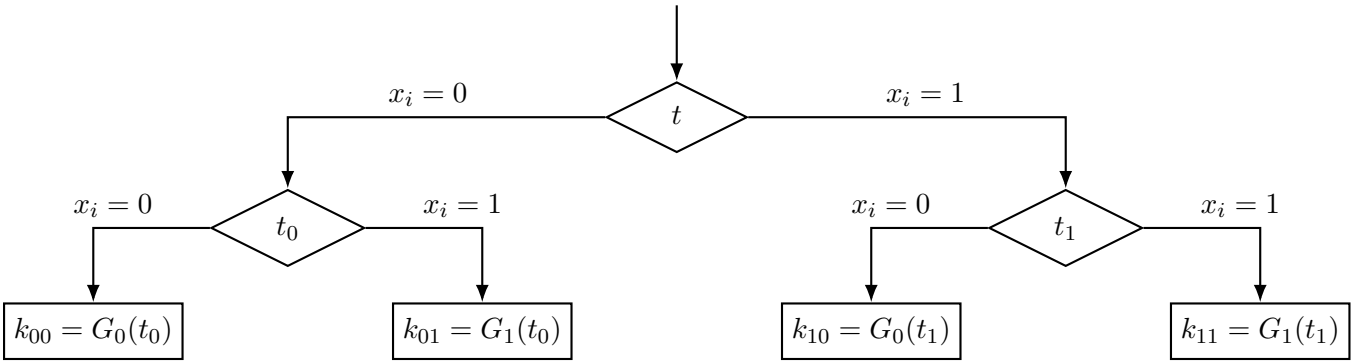
$$\left|\Pr[D \text{ wins } \mathsf{GuessGame}] - \frac{1}{2}\right| > \varepsilon(n)$$

In other words, D can distinguish between the output of a PRG and a random function. Now let us construct hybrids H_i for $1 \leq i \leq d$ where d is the depth of the decision tree mentioned before. Each hybrid H_i is a distribution of outputs of some function F defined as follows. Let $k \in \{0,1\}^i$ such that $k = x_i x_{i-1} \dots x_1$. Let us define a table T which we initally have empty and we will use to construct a truly random function in polynomial time as we have discussed previously. This table T will map i bits to n bits. Then we have:

$$F_k(x) = G_{x_n}(G_{x_{n-1}}(\dots(G_{x_{i+1}}(T[k]))))$$

That notation looks pretty messy, so let's investigate it piece by piece. First, let us understand what is meant by $T[k]$. T is, as was previously stated, a table construction of a random function which can be queried in polynomial time. $T[k] = T[x_i x_{i-1} \dots x_1]$, so effectively it's just assigning a random string of length $|n|$ to the bit string $x_i x_{i-1} \dots x_1$.

Now we take the result of T_k and apply the choosing functions of G several times. What does this all mean? Well consider the tree that we constructed previously. Then, in experiment H_i, for all rows in the tree up to the ith row, the function F returns truly random values. Starting from the $(i+1)$th row, the function returns pseudorandom values. Consider the following tree where we are observing the experiment H_1 when $n = 2$. Note: consider the following notation to mean the same thing: $T[k] = t_k$.



Observe that for the 0th row and the 1st row, we have truly random values from the table T and starting on the 2nd row, we compute pseudorandom values from the generator G. Furthermore, observe that for experiment H_0, we compute only pseudorandom values and for experiment H_n we return only truly random values. Thus, by our contrary assumption, distinguisher D can distinguish between H_0 and H_n with noticeable probability ε. Therefore, by the Hybrid Lemma, $\exists i \leq n \in \mathbb{N}$ such that D distinguishes H_i and H_{i+1} with probability $\geq \frac{\varepsilon}{n}$. Note that the only difference between H_i and H_{i+1} is that in H_i, the $(i+1)$th level is computed from a PRG while in H_{i+1} the $(i+1)$th level is computed from a random function.

Observe here that the distinguisher D, in order to remain efficient, is bounded in the number of queries she can make by some polynomial $q(n)$. Since this is the case, at the ith level of the tree, the number of entries that will have been generated over the course of the experiment is $\leq q(n)$. Since there are at most polynomially many entries on the ith level, we may construct a hybrid on that level. Let us call this set of hybrids O and suppose there are k such hybrids. We define $O_0 := H_i$ and $O_k := H_{i+1}$. O_j answers the first j new queries to F using H_i and the remaining queries using H_{i+1}. So we know that D distinguishes between O_0 and O_n with probability $\frac{\varepsilon}{n}$. Therefore, by the Hybrid Lemma, $\exists j \leq k \in \mathbb{N}$ such that D distinguishes O_j and O_{j+1} with probability $\frac{\varepsilon}{nk}$.

Now, using D, we construct the following distinguisher $\mathcal{A}$ which distinguishes PRGs from randomly chosen values:

1. On input x which is an n-bit output of either a PRG or a randomly chosen value, replace the $j+1$ entry of level i in the tree with x.
2. Ask D whether the current experiment is O_j or O_{j+1}:
   (a) If D says O_j, output that x came from a PRG.
   (b) Else, if D says O_{j+1}, output that x was randomly sampled.

$\mathcal{A}$ succeeds in distinguishing x with the same probability that D succeed in distinguishing O_j and O_{j+1} which is $\frac{\varepsilon}{kn}$. That is, $\mathcal{A}$ succeeds with noticeable probability which violates the pseudorandom property of the PRG. This is a contradiction. So we must conclude, finally, that F is indeed pseudorandom.

■