Lecture 4: Pseudorandomness - II

Last Time

- Hard Core Predicates
- Computational Indistinguishability
- Prediction Advantage
- Pseudorandom Distributions & Next-bit Unpredictability

Today

- Completeness of Next-bit Test for Pseudorandomness
- Pseudorandom Generators
  - 1-bit stretch
  - Polynomial stretch
- Pseudorandom functions

Recall

Definition (Pseudorandom Ensembles)

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

$$\{X_n\} \approx \{U_{\ell(n)}\}$$

Definition (Next-bit Unpredictability)

An ensemble of distributions $\{X_n\}$ over $\{0,1\}^{\ell(n)}$ is next-bit unpredictable if, for all $0 \leqslant i < \ell(n)$ and n.u. PPT $\mathcal{A}$, $\exists$ negligible function $\nu(\cdot)$ s.t.:

$$\Pr[t = t_1 \dots t_{\ell(n)} \sim X_n \colon \mathcal{A}(t_1 \dots t_i) = t_{i+1}] \leqslant \frac{1}{2} + \nu(n)$$

Theorem (Completeness of Next-bit Test)

If $\{X_n\}$ is next-bit unpredictable then $\{X_n\}$ is pseudorandom.

Next-bit Unpredictability $\implies$ Pseudorandomness

$$H_n^{(i)} := \left\{x \sim X_n, u \sim U_n \colon x_1 \dots x_i u_{i+1} \dots u_{\ell(n)}\right\}$$

- First Hybrid: H_n^0 is the uniform distribution $U_{\ell(n)}$
- Last Hybrid: $H_n^{\ell(n)}$ is the distribution X_n
- Suppose $H_n^{(\ell(n))}$ is next-bit unpredictable but not pseudorandom
- $H_n^{(0)} \not\approx H_n^{(\ell(n))} \implies \exists\, i \in [\ell(n) - 1]$ s.t. $H_n^{(i)} \not\approx H_n^{(i+1)}$
- Now, next bit unpredictability is violated
- Exercise: Do the full formal proof

Pseudorandom Generators (PRG)

Definition (Pseudorandom Generator)

A deterministic algorithm G is called a *pseudorandom generator* (PRG) if:

- G can be computed in polynomial time
- $|G(x)| > |x|$
- $\left\{x \leftarrow \{0,1\}^n : G(x)\right\} \approx_c \left\{U_{\ell(n)}\right\}$ where $\ell(n) = |G(0^n)|$

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

- Can we construct PRG with even 1-bit stretch?
- What about many bits? Can we generically stretch?

PRG with 1-bit stretch

- Remember the hardcore predicate?
- It is hard to guess $h(s)$ even given $f(s)$
- Let $G(s) = f(s) \| h(s)$ where f is a OWF
- Some small issues:
  - $|f(s)|$ might be less than $|s|$
  - $f(s)$ may always start with prefix 101 (not random)
- **Solution:** let f be a one-way *permutation* (OWP) over $\{0,1\}^n$
  - Domain and Range are of same size, i.e., $|f(s)| = |s| = n$
  - $f(s)$ is uniformly random over $\{0,1\}^n$ if s is

    $\forall y : \Pr[f(s) = y] = \Pr[s = f^{-1}(y)] = 2^{-n}$

    $\Rightarrow f(s)$ is uniform and cannot start with a fix value!

PRG with 1-bit stretch

- Let $f : \{0,1\}^* \to \{0,1\}^*$ be a **OWP**
- Let $h : \{0,1\}^* \to \{0,1\}$ be a hardcore predicate for f
- **Construction:** $G(s) = f(s) \parallel h(s)$

Theorem (PRG based on OWP)

G is a pseudorandom generator with 1-bit stretch.

- Think: Proof?
- Proof Idea: Use next-bit unpredictability. Since first n bits of the output are uniformly distributed (since f is a permutation), any adversary for next-bit unpredictability with non-negligible advantage $\frac{1}{p(n)}$ must be predicting the $(n+1)$th bit with advantage $\frac{1}{p(n)}$. Build an adversary for hard-core predicate to get a contradiction.

One-bit stretch PRG $\implies$ Poly-stretch PRG

Intuition: Iterate the one-bit stretch PRG poly times

Construction of $G_{poly} : \{0,1\}^n \to \{0,1\}^{\ell(n)}$:

- Let $G : \{0,1\}^n \to \{0,1\}^{n+1}$ be a one-bit stretch PRG

$$\begin{array}{rcl} s & = & X_0 \\ G(X_0) & = & X_1 \| b_1 \\ & \vdots & \\ G(X_{\ell(n)-1}) & = & X_{\ell(n)} \| b_{\ell(n)} \end{array}$$

- $G_{poly}(s) := b_1 \ldots b_{\ell(n)}$

Think: Proof?

Proof that G_{poly} is pseudorandom

- Want: $\left\{s \leftarrow \{0,1\}^n : G_{poly}(s)\right\} \approx_c \left\{U_{\ell(n)}\right\}$
- Let D be any non-uniform PPT algorithm.

Step 0:

$$\begin{array}{rcl} \multicolumn{3}{c}{\underline{\text{Experiment } H_0}} \\ s & = & X_0 \\ G(X_0) & = & X_1 \| b_1 \\ G(X_1) & = & X_2 \| b_2 \\ & \vdots & \\ G(X_{\ell-1}) & = & X_\ell \| b_\ell \end{array}$$

Output $D(b_1 b_2 \ldots b_\ell)$

Claim: $\left| \Pr_s[D(G'(s)) = 1] - \Pr_s[H_0 = 1] \right| = 0.$

Proof: Input of D is identically distributed in both cases. $\square$

Proof that G_{poly} is pseudorandom

Step 1: modify H_0 one line at a time.

$$\begin{array}{rcl}
\multicolumn{3}{c}{\underline{\text{Experiment } H_0}} \\
s & = & X_0 \\
G(X_0) & = & X_1 \| b_1 \\
G(X_1) & = & X_2 \| b_2 \\
& \vdots & \\
G(X_{\ell-1}) & = & X_\ell \| b_\ell
\end{array}$$

Output $D(b_1 b_2 \ldots b_\ell)$.

Proof that G_{poly} is pseudorandom

Step 1: modify H_0 one line at a time.

Experiment H_0	Experiment H_1
$s = X_0$	$s = X_0$
$G(X_0) = X_1 \| b_1$	$X_1 \| b_1 = s_1 \| u_1$
$G(X_1) = X_2 \| b_2$	$G(s_1) = X_2 \| b_2$
$\vdots$	$\vdots$
$G(X_{\ell-1}) = X_\ell \| b_\ell$	$G(X_{\ell-1}) = X_\ell \| b_\ell$
Output $D(b_1 b_2 \ldots b_\ell)$.	Output $D(u_1 b_2 \ldots b_\ell)$.

Claim: $\left| \Pr_s[H_0 = 1] - \Pr_{s,s_1,u_1}[H_1 = 1] \right| \leqslant \mu(n)$

- Can similarly define $H_2, \ldots, H_{\ell-1}$ s.t. in $H_{\ell-1}$, $b_1 b_2 \ldots b_\ell$ is sampled from U_ℓ
- To prove that G_{poly} is PRG, it suffices to show that $H_0 \approx_c H_\ell$

Proof that G_{poly} is pseudorandom (contd.)

Step 2: Hybrid Lemma

- For contradiction, suppose that G_{poly} is not a PRG, i.e., H_0 and H_ℓ are distinguishable with non-negligible probability $\frac{1}{p(n)}$
- By Hybrid Lemma, there exists i s.t. H_i and H_{i+1} are distinguishable with probability $\frac{1}{p(n)\ell(n)}$
- <u>Idea</u>: Contradict the security of G

Proof that G_{poly} is pseudorandom (contd.)

Step 3: Breaking security of G

- For simplicity, suppose that $i = 0$ (proof works for any i)
- Construct D to break the pseudorandomness of G as follows
  - D gets as input $Z \| r$ sampled either as $X_1 \| b_1$ or as $s_1 \| u_1$
  - Compute $X_2 \| b_2 = G(Z)$ and continue as the rest of the experiment(s)
  - Output $D(rb_2 \ldots b_\ell)$
- If $Z \| r$ is pseudorandom, i.e., sampled as $X_1 \| b_1 = G(s)$, then output of D is distributed identically to the output of H_0
- Otherwise, i.e., $Z \| r$ is (truly) random, and therefore output of D is is distributed identically to the output of H_1
- Hence: D distinguishes the output of G with advantage $\frac{1}{p(n)\ell(n)}$ and runs in polynomial time. This is a contradiction $\square$

Concluding Remarks on PRG

- OWF $\implies$ PRG: [Impagliazzo-Levin-Luby-89] and [Hastad-90]
  - Celebrated result! Good to read
- More Efficient Constructions: [Vadhan-Zheng-12]
- Computational analogues of Entropy
- Non-cryptographic PRGs and Derandomization: [Nisan-Wigderson-88]

Going beyond Poly Stretch

- PRGs can only generate polynomially long pseudorandom strings
- Think: How to efficiently generate exponentially long pseudorandom strings?

Idea: Functions that index exponentially long pseudorandom strings

Random Functions

- How do we define a random function?
- Consider functions $F : \{0,1\}^n \to \{0,1\}^n$
- Think: How many such functions are there?
- Write F as a table:
  - first column has input strings from 0^n to 1^n;
  - against each input, second column has the function value
  - i.e., each row is of the form $(x, F(x))$
- The size of the table for $F = 2^n \times n = n2^n$
- Total number of functions mapping n bits to n bits $= 2^{n2^n}$

Random Functions

There are two ways to define a random function:

- **First method**: A random function F from n bits to n bits is a function selected *uniformly at random* from all 2^{n2^n} functions that map n bits to n bits
- **Second method:** Use a randomized algorithm to describe the function. Sometimes more convenient to use in proofs
  - randomized program M to implement a random function F
  - M keeps a table T that is initially empty.
  - on input x, M has not seen x before, choose a random string y and add the entry (x, y) to the table T
  - otherwise, if x is already in the table, M picks the entry corresponding to x from T, and outputs that
- M's output distribution identical to that of F.

Random Functions

- Truly random functions are huge random objects
- No matter which method we use, we cannot store the entire function efficiently
- With the second method, we can support **polynomial** calls to the function efficiently because M will only need polynomial space and time to store and query T
- Can we use some crypto magic to make a function F' so that:
  - it "looks like" a random function
  - but actually needs much fewer bits to describe/store/query?

Pseudorandom Functions (PRF)

- PRF looks like a random function, and needs polynomial bits to be described
- Think: What does "looks like" mean?
- First Idea: Use computational indistinguishability
  - Random Functions and PRFs are hard to tell apart efficiently
- Think: Should the distinguisher get the *description* of either a random function or a PRF?
- **Main Issue:** A random function is of exponential size
  - D can't even read the input efficiently
  - D can tell by looking at the size
- **Idea**: D can only *query* the function on inputs of its choice, and see the output.