CS 830: Randomness and Complexity 2/15/05

Lecture 9: One-way Functions to Pseudorandom Generators

instructor: Jin-Yi Cai Scribe: Matthew Anderson

In this lecture we discuss using one-way functions to construct pseudorandom generators.

1 The Discrete Log

1.1 Background

Blum and Micali presented the construction a pseudorandom generator using the hardness of the discrete log problem, which can be formalized as:

$$f : x \mapsto g^x (\text{mod } p), 1 \leq x \leq p-1, \tag{1}$$

for some prime p, and g a generator of $\mathbf{Z_p}$. f is a one-way permutation. Given g^x the task is to calculate x. This is believed to be a hard problem. Finding the least significant bit of x is easy, it is equivalent to determining if g^x is a quadratic residue. It can be shown that if f is a one-way function then finding the leading (most significant) bit of x is hard. This one-way function can then be used to construct a next-bit unpredictable generator, which will be discussed shortly.

Yao, soon after, defined the term "pseudorandomness" which gave a more general model of randomness than the statistical model presented by Knuth. Yao showed that his definition of "pseudorandomness" was equivalent to the next-bit unpredictable model presented by Blum and Micali. He also showed that probabilistic polytime algorithms could be simulated in sub-exponential time.

Blum and Micali's result showed how to construct a pseudorandom generator from a one-permutation. Others have since shown how to construction pseudorandom generators from sequentially more general one-way functions:

- one-to-one one-way functions.
- k-to-one one-way functions, for known constant k.
- general one-way functions.

This last step of the work was completed by Hastab, Impagliazzo, Levin and Luby.

1.2 Constructing a Pseudorandom Generator from the Discrete Log Problem

The quadratic residuosity of a number y:

$$QR(y \text{ mod } p) = \left(\frac{y}{p}\right) \tag{2}$$

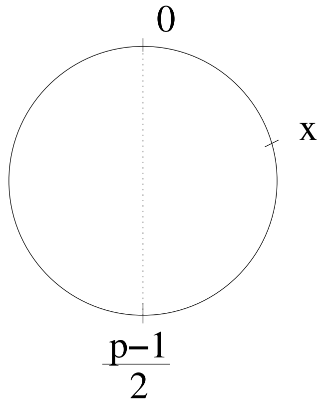

Figure 1: The group $\mathbf{Z_p}$ represented as a circle.

where $\left(\frac{y}{p}\right)$ is the Legendre symbol. If a number is a quadratic residue we can compute its two square roots, one positive and one negative. We saw last time that if $p \equiv 3(\text{mod}4)$ then this was easy. In general, the square roots can be computed using a polynomial time randomized algorithm. Define the function:

$$B_{g,p}(y) = \left\{ \begin{array}{ccc} 1 & \log_{g,p} y < \frac{p-1}{2} & x \text{ is in first half} \\ 0 & \text{otherwise} & x \text{ is in second half} \end{array} \right\} \tag{3}$$

First, we will try to get a better understanding of how B works. Consider $y = g^x$. We can represent the elements of the group $\mathbf{Z_p}$ as a circle. The circle can be partitioned into two sections, the elements from zero to $\frac{p-1}{2} - 1$ and the elements greater than or equal to $\frac{p-1}{2}$, this is shown in Figure 1.

$B(y)$ can determine the most significant bit of x. We will show that finding the discrete log can be reduced to finding the most significant bit of x. B can also be used to the determined which of the two roots of a square is the principle one (the principle root has an exponent less than $\frac{p-1}{2}$). The idea is as follows, if the least significant bit of x is a zero then g^x is a quadratic residue, otherwise g^x is a quadratic non-residue. The quadratic residue determination gives the least significant bit of x. If g^x is a non-residue, then $g^x g^{-1}$ is a quadratic residue. Having converted the non-residues to residues we can take the square roots using our randomized polytime algorithm. We get two roots $r_1 = r$ and $r_2 = rg^{\frac{p-1}{2}}$. We can repeat this process until we have "sucked" all the bits of x out of g^x.

More formal the algorithm is:

(1) Input $y = g^x$, let $elt := y$, $ind = \epsilon$.
(2) if ($elt = 1$) return ind.
(3) Test $QR(elt)$: if (residue) $ind := 0ind$, goto 5 else $ind := 1ind$, goto 4.
(4) $elt := elt \cdot g^{-1}$ (change exponent to exponent - 1)
(5) Find the two square roots of elt, use $B(elt)$ to select the principle square root and set $elt :=$ principle root. goto 2.

So given a blackbox for determining the most significant bit of x, $B(y)$, we can compute the discrete log of y efficiently. So if discrete log is hard, then finding the most significant bit is also hard.

The Blum-Micali generator uses the fact that discrete log is thought to be hard. Given an input x chosen uniformly at random from $\{1, ..., p-1\}$ we can construct a sequence of pseudorandom bits.

$$
\begin{array}{lllll}
x_0 = x & \mapsto x_1 = g^{x_0} & \mapsto x_2 = g^{x_1} & \mapsto \ldots \mapsto x_i = g^{x_{i-1}} \bmod p & \mapsto \ldots \\
\vdots & \vdots & \vdots & \vdots & \\
b_0 & b_1 & b_2 & b_i = B(x_i) &
\end{array} \tag{4}
$$

We can use this method to compute a polynomial number of pseudorandom bits, $b = (b_{poly(n)} \ldots b_i \ldots b_0)$, from our initial random seed of x. Given an x_i it is easy to generate, the bits $(b_{poly(n)} \ldots b_{i+1})$, because you simply apply the original process. However, you cannot compute b_i easily, knowing only x_i because of the difficulty of the discrete log problem. Thus, this generator is next-bit unpredictable. Given the bits $b_0 b_1 \ldots b_i$ it is still hard to compute the b_{i+1} bit, because you are only given the most significant bits.

Homework: (Random Self Reducibility) Suppose you could construct a $\tilde{B}(y)$ with $1/2 + \epsilon$ probability of correctly computing $B(y)$, where $\epsilon \sim \frac{1}{\text{poly}}$. Given $y = g^x$, compute x in randomized polynomial time. [Hint: Thinking in the circle representation discussed before, you can "spin" x around by multiplying $y' = y \cdot g^r$, for some random known r. You can then apply $\tilde{B}(y')$ to determine the region of $x + r$. You can repeat this process a polynomial number of times until the probability of x being a certain value is sufficiently high.]

2 Hardcore Bit

Given a one-way permutation $f : x \mapsto f(x)$ we can construct pseudorandom generator as follows:

$$
\begin{aligned}
(x, y) &\mapsto (f(x), y, x \cdot y) \\
(f(x), y, x \cdot y) &\mapsto (f(f(x)), y, f(x) \cdot y, x \cdot y) \\
&\vdots
\end{aligned} \tag{5}
$$

where x and y are uniformly chosen from $\{0,1\}^n$, and $(\cdot)$ is the inner product (dot product: $\{0,1\}^n \times \{0,1\}^n \to \{0,1\}$). If f is a one-way permutation, the output of the this process is a sequence of $poly(n)$ pseudorandom bits, $(f^{poly(n)}(x) \cdot y, \ldots, f^2(x) \cdot y, f(x) \cdot y, x \cdot y)$.

You can also improve the inner product method by by using matrices instead of vectors,

$$
\begin{aligned}
(x, y) &\mapsto (f(x), y, x \cdot y) \\
n + rn &\mapsto n + rn + r
\end{aligned} \tag{6}
$$

where r is the number of columns in the public random matrix y. We will discuss this further next lecture.

Suppose we do not have a one-way permutation, and instead we have a more general one-to-one one-way function, g.

$$
g(x) \mapsto \text{ extract out } n - \epsilon \text{ bits} \tag{7}
$$

Then you take a hash function, h, called an "extractor", which when applied to $g(x)$ is ϵ away from $U_{n-\epsilon}$. Suppose we instead have a k-to-one one-way function for some fixed k, the computation becomes even messier because we must also hash y as well as x and measure the entropy of the output.

$$\begin{aligned} x &\mapsto f(x), h_{y_1}(x) \\ y &\mapsto h_{y_1}(f(x)) \\ z &\mapsto x \cdot z \end{aligned} \tag{8}$$

Homework: Show that given a pseudorandom generator, one can construct a one-way function.

Research problem: (Cryptographic one-way functions) The set of languages in UP are the set languages in NP with exactly one accepting path. Is QR in UP? Is QNR in UP? Show, given f is a one-to-one function and f is in P and f^{-1} is not P that such an f exists if and only if UP != P.

Suppose you have a pseudorandom generator that takes $\log^2 n$ bits $\mapsto n^k$ bits, show $BPP \subseteq DTIME[n^{O(\log n)}]$.

What pseudorandom generator will give $BPP \subseteq \bigcap_{\epsilon>0} DTIME[2^{n^\epsilon}]$?

What strength of pseudorandom generator will give $BPP \subseteq \bigcup_{k>0} DTIME[2^{(\log n)^k}]$?

Next lecture we will further discuss the improvement on the inner product generator by using public random matrices and prove its hardness.