# Notes for Lectures 3–5

## 1 Definition of next-bit-unpredictability

Instead of having truly random one-time pads, we'll try for pseudorandom ones. We will postpone the harder question of how to encrypt securely and will focus only on obtaining long random strings for now.

Our first definition of pseudorandomness (below), due to Blum and Micali and first published in 1982, will capture the following feature of truly random strings: you can't predict the next bit, even given all the previous ones. Except that we will limit the computational power of our bit predictor to probabilistic polynomial time.

**Definition 1.** A bit predictor $A$ is a probabilistic polynomial-time interactive algorithm. It runs in stages. At first, $A$ receives $1^k$ as input (for some $k$). At the end of each stage, $A$ can output `next` or a bit $b$. If a stage outputs `next`, $A$ expects one more bit of input, and enters the next stage. If a stage outputs $b$, then $A$ is finished, and $b$ is called the output of $A$.

Note that giving $A$ input $1^k$ ($k$ written in unary) instead of $k$ allows $A$ to have running time polynomial in the length of $|1^k|$, which is $k$, rather than in the length of binary representation of $k$, which is merely $\log k$. This is simply a technical notational trick.

Let $G : \{0,1\}^* \to \{0,1\}^*$ be a polynomial-time deterministic algorithm. Supppose $|G(x)| = l(|x|)$ for some *expansion* function $l$ satisfying $l(k) > k$. Let $A$ be a bit predictor. Consider the following experiment `experiment-predict`, parameterized by $k$:

1. Select a random $x$ of length $k$
2. Compute $y = G(x)$
3. Run $A(1^k)$, giving it bits of $y$ in order in response to $A$'s `next` requests

If $A$ stops after $i \le l(k)$ stages and outputs $b = y_i$, we say that `experiment-predict` *succeeds*. We will define a pseudorandom generator as an algorithm $G$ for which $A$ cannot succeed with all but a negligible advantage (over the probability 1/2 of random guessing). First we define negligible.

**Definition 2.** A nonnegative function $f : \mathbb{N} \to \mathbb{R}$ is *negligible* if, for any positivie polynomial $p$, $f \in o(1/p)$.

Note trivially that if $f(k)$ is negligible, so is $f(ck)$, $cf(k)$, $f(k^c)$ and $f(k)^c$ for any constant $c$. We will use these facts in most future proofs.

**Definition 3 ([BM84]).** $G$ is a pseudorandom generator if for each bit predictor $A$, there exists a negligible function $\mathrm{negl}(k)$ such that for all $k$,

$$Pr[\texttt{experiment-predict}(k) \text{ succeeds}] \le 1/2 + \mathrm{negl}(k).$$

## 2 Discrete Logarithm Assumption

Recall that modulo every prime $p$, there exists a generator $g$ of $\mathbb{Z}_p^*$, i.e., $g$ such that $g, g^2 \bmod p, g^3 \bmod p, \ldots, g^{p-1} \bmod p$ actually covers the whole set $\{1, 2, \ldots, p-1\}$. Moreover, there exists an algorithm that, for any $k$, finds a random $k$-bit prime $p$ and some generator $g$ of $\mathbb{Z}_p^*$ in time polynomial in $k$.

Notation for the purposes of this course: $\mathbb{Z}_p = \{0, 1, ..., p-1\}$ and $\mathbb{Z}_p^* = \{1, 2, ..., p-1\}$. $\mathbb{Z}_p^*$ is generally understood to mean that we are interested in the multiplication modulo $p$ operation.

**Assumption 1.** For any poly-time algorithm $A$, there exists a negligible function negl such that, if you generate random $k$-bit $p$ and its generator $g$ (according to the algorithm described above) and select a random $x \in \mathbb{Z}_p^*$, $\Pr[A(p, g, g^x \bmod p) = x] \leq \text{negl}(k)$.

## 3 First attempt at a PRG

Now, let's try to build a generator based on that assumption. Select a random $p$ of length $k$, generator $g$ of $\mathbb{Z}_p^*$, and $x \in \mathbb{Z}_p^*$. For some $n$ (greater that the number of random bits you used to generate $p, g, x$) , compute $x_1 = x$, $x_2 = g^{x_1}$, $x_3 = g^{x_2}, \ldots, x_n = g^{x_{n-1}}$ (all modulo $p$). Output, in reverse order, $x_n, x_{n-1}, x_{n-2}, \ldots, x_1$. Seems like should be hard to predict (because you need to take DL of $x_i$ to get $x_{i-1}$), but clearly something is fishy, because we know from PS2 that given $g^x$, you can tell whether $x$ has last bit 0 or 1 (i.e., $x$ is a square or not), so clearly you can predict some bit of $x_{i-1}$ given $x_i$.

How do we fix this?

## 4 Blum-Micali PRG

The following construction was proposed together with the above definition of unpredictability in [BM84].

Define $B(x) = \{0$, if $x < p/2$, and 1 otherwise $\}$.

**Lemma 1.** *Given $g^x$, $B(x)$ is unpredictable. I.e., for every polynomial-time algorithm $D$ there exists a negligible function* negl$(k)$ *such that for all $k$,*

$$\Pr[D(p, g, g^x \bmod p) = B(x)] \leq 1/2 + \text{negl}(k),$$

*where $p$ is a random $k$-bit prime, $g$ generator, $x \in_R \mathbb{Z}_p^*$.*

Using this lemma, we can prove the following theorem.

**Theorem 1.** *If, instead of outputting $x_n, x_{n-1}, x_{n-2}, \ldots, x_1$ we output $B(x_n), B(x_{n-1}), B(x_{n-2}), \ldots, B(x_1)$, we get a PRG as defined above, as long as the DL assumptions holds.*

We will prove the theorem first, then the lemma.

**Proof of Theorem 1** Suppose it's not a PRG. Then there exists a bit predictor $A$.

How will we prove the lemma? By *reduction* (note that this is the first reduction proof we will see!). That is, from such an evil $A$ that violates pseudorandomness, we will build $D$ that violates claim 1. Hence, it will be a contradiction.

So, now, let's build.

First of all, as input we are given $p, g, y = g^x$, and we have to predict $B(x)$. We can use $A$ to help.

Notice that $A$ has to succeed in predicting some bit: first, second, ..., $i$-th, ..., $n$-th. Pick $i$ at random between 1 and $n$.

Set $x_{n-i+2} = y$ and compute $x_{n-i+3} = g^{x_{n-i+2}}$, $x_{n-i+4} = g^{x_{n-i+3}}, \ldots, x_n = g^{x_{n-1}}$. When $A$ asks for the first bit, give it $B(x_n)$; for the next bit, give it $B(x_{n-1})$, and so on. Note that $A$ is getting the same asnwers as it would be getting when $G$ is run on $x_1 = DL(DL(DL(...(x)...)))$ (there are $n - i$ DLs here); so the distribution of the inputs to $A$ is the same as in `experiment-predict`, so $A$ would have the same probability of success.

Recall that $A$ guesses some bit of the string. Because we chose $i$ at random, $A$ has $1/n$ probability of trying guessing exactly the $i$-th bit, which is $B(x_{n-i+1}) = B(x)$, which is what we need. So in $1/n$ of the cases, we will use $A$'s guess. Else (if $A$ guesses before the $i$-th bit, or asks for the $i$-th bit which we don't know), we simply guess a bit $b$ at random.

Suppose $A$ has probability of $1/2 + \epsilon$ of being correct. Then what is the probability of $D$ being correct? It's $(1/2)(n-1)/n + (1/n)(1/2 + \epsilon) = 1/2 + \epsilon/n$. So if $A$'s advantage over 1/2 is not negligible, neither is $D$'s. Contradiction.

**Proof of Lemma 1** Suppose there is a predictor. Then we'll build a machine that takes discrete logs. Recall from PS2, there are two square roots of $g^x$, if $x$ is even. One is $r_1 = g^{x/2}$, and the other is $r_2 = g^{x/2+(p-1)/2}$. Note that $B(x/2) = 0$ and $B(x/2 + (p-1)/2) = 1$, so if we have a predictor $D$, then we can distinguish $r_1$ from $r_2$. So here's the algorithm for taking discrete logarithms using the bit predictor: consider $y = g^x$. Find the last bit of $x$ (by simply seeing if $y$ is a square or not by raising it to $(p-1)/2$). If it's 1, divide $y$ by $g$ to make it zero. Now take the square root of $y$ modulo $p$ (there are efficient algorithms for it). You have two roots $r_1$ and $r_2$. Get $r_1$ (by using $D$). Thus, $r_1$ is the same as $y$, except $x$ is right-shifted by one bit. And you know the bit that came out. Repeat, until you get all of $x$ bit-by-bit.

What's wrong with this proof? It only works if $D$ is perfect. It's more complicated if $D$ only succeeds sometimes. We won't do it in class; see [BM84].

## Discussion

This pseudorandom generator requires one modular exponentiation for each output bit. Naive algorithms for modular exponentitation take $\Theta(k^3)$ bit operations (where $k$ the length of the modulus and the exponent); more sophisticated algorithms can do only slightly better. Common values for $|p|$ given the strength of today's discrete logarithm algorithms are somewhere in the range of 1,024–2,048 bits; a single pseudorandom bit would take a few milliseconds to compute on today's PCs.

Besides its relatively low speed, another disadvantage of this PRG is that one needs to know the number $n$ of output bits in advance (because bits are output backwards). We will fix it in the next couple of lectures.

Finally, recall that pseudorandom generators were defined to take truly random strings as inputs, rather than primes, generators, etc. This minor technical point can be rectified in one of two ways. First, we can redefine the PRG to work with "public values" (in this case, $p$ and $g$) that are known to everyone, including the bit-predictor. Then $x$, which is a truly random seed, is the only input remaining. Alternatively, we can redefine the PRG to take a truly random string as input, and use it for generating a random $p$, a random $g$ and a random $x$ before starting the actual bit generation.

## References

[BM84] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM Journal on Computing*, 13(4):850–863, November 1984.