2 Pseudorandom Generators: Unpredictability and First Example

2.1 Definition of next-bit-unpredictability

As we have seen, information-theoretic security requires long random strings. This brings up the following question: can we replace random strings with pseudorandom ones and still retain some notion of security? For now, we will focus on pseudorandomness and postpone the question of what notion of security we can achieve using it. It will turn out that understanding pseudorandomness well will be of great help for understanding secure encryption.

A common understanding of the meaning of "pseudorandom" is something that looks random but is generated by a deterministic process starting with a random seed. The meaning "looks random" can vary, and is crucial to the definition. 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. The definition will require pseudorandom strings to have this property when the computational power of the bit predictor is limited: we will limit the predictor's expected running time to some polynomial in the length of the input seed.

We will define the bit predictor as an algorithm that reads one bit of the pseudorandom string at a time, and each time decides whether to try to predict the next bit, or to read it. As initial input, it will receive the length k of the random seed used to generate the string (but not the seed itself, which must remain secret for unpredictability). For technical reasons to be explained shortly, the value k will be input in unary as a string of k ones, denoted 1^k.

Definition 1. A bit predictor A is an algorithm that 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.

We will now formally define how a bit predictor interacts with a potential pseudorandom generator G. Let $G : \{0,1\}^* \to \{0,1\}^*$ be a polynomial-time deterministic algorithm. Suppose the length of the output of G is always greater than the length of the input, and furthermore the length of the output is the same as long as the length of the input is the same: $|G(x)| = l(|x|)$ 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 $i \leq l(k)$ stages and outputs $b = y_i$, we say that `experiment-predict` *succeeds*.

We would like to say that G is pseudorandom if prediction succeeds with probability no better than 1/2 (clearly, any predictor can get 1/2 by ignoring its inputs and flipping a random coin). However, that's too much to ask from a pseudorandom generator. For instance, if the predictor tries to guess the seed x, then it can check if the bits it is getting match $G(x)$ until it is reasonably sure that the guess for x is correct, and the simply compute the correct value for the next bit. Of course, the chances of the correctness of the guess are tiny (2^{-k}, to be precise), but nonetheless they raise the probability of the predictor's success above 1/2.

Thus, for functions $t(k)$ and $\epsilon(k)$, we will say that G is (t, ϵ)-unpredictable if no adversary with running time[1] $t(k)$ has can succeed in `experiment-predict` with probability better that $1/2 + \epsilon(k)$ (observe that "more unpredictable" would mean larger t and smaller ϵ). This definition is quite precise because it keeps

[1]We must include description size in some canonical language into the running time; alternatively, we can talk of $t(k)$ as the circuit size of A.

track of the resources and success rates of the adversary, but it leads to many details to keep track of in proofs. We will instead use a more coarse definition, even though it is less informative, because it is a little bit easier to work with, especially when you are doing cryptography for the first time. We will simply say that we are happy as long as for any polynomial $t(k)$, there is a negligible function $\epsilon(k)$ such that G is (t, ϵ) unpredictable. It is important to note that all the results in this class can be reworked to obtain concrete values of (t, ϵ) (which are important in practice, when you have concrete adversaries to protect against) if one cares to simply follow the details.

First we define negligible.

Definition 2. A nonnegative function $f : \mathbb{N} \to \mathbb{R}$ is *negligible* if, for any positive 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 $\eta(k)$ such that for all k,

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

Now we are ready to explain why A gets 1^k rather than k as input. This 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, needed because historically the term "polynomial-time algorithm" means "polynomial in the input length," not "polynomial in the input value": we have to make the input length equal to k.

2.2 On the Necessity of Assumptions

When one tries to build pseudorandom generators, one runs into the following problem: predicting the next bit is in NP. Because we don't know that P is not equal to NP, if P=NP, then one can alway predict the next bit in polynomial time, thus making pseudorandom generators impossible to build.

More formally, suppose the adversary is given bits $y_1 \dots y_{i-1}$ and has to predict y_i. Since the adversary knows G, the adversary can build a circuit C that is satisfied only by inputs x for which the first $i-1$ bits of $G(x)$ are $y_1 \dots y_{i-1}$. Furthermore, the adversary can modify that circuit to make C_1, which satisfied only by inputs x for which the first $i-1$ bits of $G(x)$ are $y_1 \dots y_{i-1}$ and for which the i-th bit of $G(x)$ is 1. Similarly, the adversary can modify C to make C_0, which satisfied only by inputs x for which the first $i-1$ bits of $G(x)$ are $y_1 \dots y_{i-1}$ and for which the i-th bit of $G(x)$ is 0. (Thus, $C(x)$ is the "or" of $C_0(x)$ and $C_1(x)$.) Then, if C_0 is satisfiable and C_1 is not, $y_i = 0$ is the correct prediction; else if C_1 is satisfiable and C_0 is not, $y_i = 1$ is the correct prediction (it could be that both are satisfiable, which means x is not unique and both predictions are possible).

If P=NP, then circuit satisfiability can be decided in polynomial time, and we can build a bit predictor as follows. Request bits until exactly one of C_0 and C_1 is satisfiable (this must happen, because the length $l(k)$ of the pseudorandom string y is greater than the length k of the seed x, so not all strings y are possible, so at some point only a single possibility for the next bit must remain), and then predict the bit according to whichever circuit is satisfiable. This predictor will be always correct!

Therefore, at the very least we must assume that P≠NP in order to build pseudorandom generators. In fact, we will have to make much stronger assumptions about intractability of particular problems in the *average* case (whereas P vs. NP question is about only the worst case).

2.3 Discrete Logarithm Assumption

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.

It is a fact from number theory 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, p and g can be found efficiently: 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.

The function $f(x) : \mathbb{Z}_p^* \to \mathbb{Z}_p^*$ that is computed as $f(x) = g^x \bmod p$ is a bijection (a.k.a. permutation) of $\mathbb{Z}_p^*$. Its inverse is known as the "discrete logarithm": when working modulo p, we will denote by $\log_g y$ the unique value x, $1 \le x \le p-1$, such that $g^x \equiv y \pmod{p}$.

As we know, f is easy to evaluate. It is believed (because many have tried and failed) that f is hard to invert. Namely, the following assumption is widely believed to hold.

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

2.4 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}). However, all that DL assumption says is that it is hard to predict *values*, not *individual bits*. In fact, we know from PS2 that given g^x, you can tell whether x has last bit 0 or 1 (i.e., g^x is a square or not), so clearly you can predict some bit of x_{i-1} given x_i. Thus, at least some bits are predictable, and hence this is not a proper PRG.

To build a PRG, we need to condense the hardness of predicting x_{i-1} down the hardness of predicting a single bit. We do this below.

2.5 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 $\eta(k)$ such that for all k,*

$$\Pr[D(p, g, g^x \bmod p) = B(x)] \le 1/2 + \eta(k),$$

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 answers as it would be getting when G is run on $x_1 = \log_g(\log_g(\log_g(...(x)...)))$ (there are $n-i$ logs here). Note also that if x is distributed uniformly, so is x_1, because modular exponentiation (and hence its inverse, discrete logarithm) is a permutation $\mathbb{Z}_p^*$. Hence, what A observes is the same as the output of the PRG on a completely random input x_1. 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 exponentiation 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 pseudo-random bits. *SIAM Journal on Computing*, 13(4):850–863, November 1984.