

BU CAS CS 538: Cryptography
Lecture Notes. Fall 2005.
<http://www.cs.bu.edu/~itkis/538/>

Gene Itkis*

Boston University Computer Science Dept.

Notes for Lectures 3–5: Pseudo-Randomness; PRGs

1 Randomness

Randomness is a complex concept. A very descent survey can be found at

<http://en.wikipedia.org/wiki/Randomness>.

For somewhat more technical discussions see the notes of Prof. Levin (and the references therein):

- <http://www.cs.bu.edu/fac/lnd/toc/z/node23.html#SECTION00063000000000000000>, and
- <http://www.cs.bu.edu/fac/lnd/expo/icm94.html>

And as long as we are discussing references on the web see

- <http://en.wikipedia.org/wiki/Pseudorandomness>,
- http://en.wikipedia.org/wiki/Pseudo-random_number_generator,
- and the more technical
<http://www.cs.bu.edu/fac/lnd/toc/z/node24.html#SECTION00064000000000000000>.

Keep in mind that while the Wikipedia references are pretty good, they are far from perfect. E.g., there, the digits of π are referred to as random. Yet, they clearly do not pass our definitions of randomness. Think about why, and about what did the author of the Wikipedia article really mean. Still, all the imprecision and defects notwithstanding, I highly recommend to read those articles and browse the references they provide.

Our guiding intuition was to treat randomness as inability to guess (with probability better than 1/2) bits that were not seen (clearly this disqualifies digits of π).

2 Pseudo-Randomness

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.

The main intuition of what 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),

* These notes are heavily based on the notes by Leo Reyzin (see <http://www.cs.bu.edu/fac/reyzin/teaching/f04cs538/index.html>), with later modifications by Gene Itkis (who is deeply grateful to Leo for kindly making the latex source for the notes available).

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 $G : \{0, 1\}^* \rightarrow \{0, 1\}^*$ 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 then 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 time¹ $t(k)$ has can succeed in `experiment-predict` with probability better than $1/2 + \epsilon(k)$ (observe that "more unpredictable" would mean larger t and smaller ϵ). This definition is quite precise because it keeps 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 $f : \mathbb{N} \rightarrow \mathbb{R}$ 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.

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 .

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*,

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

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 Alternative definitions

In class we have seen a different definition of pseudo-randomness, which we called *random-access unpredictability*².

Yet another approach (discussed below) is to define pseudo-randomness focusing on the “pseudo-”: as *indistinguishable* from random.

Which of these definitions is the “right” one to use?

Fortunately, we do not have to choose: these definitions are equivalent!

Before we show this let us formally introduce the indistinguishability-based definition of pseudorandomness.

2.3 Indistinguishability

We have seen how to build generators whose next bit is unpredictable from the previous bits. This clearly has applications: e.g., if you want to run a lottery or build a gambling machine, this is exactly what you are looking for, so that next day’s winning numbers cannot be predicted from the past. However, it is not clear that this is the right notion to apply when, for example, you want to use pseudorandom strings instead of random ones in one-time-pad encryption. For instance, if you always begin your letters with “Dear” and end them with “Sincerely,” then the adversary can figure out the first few and the last few bits of the one-time pad. In that case, you want the middle bits (which protect the actual contents of your letter) to be unpredictable to the adversary who is given some of the beginning and some of the end of the pseudorandom string. This is merely one example to illustrate that next-bit unpredictability is not necessarily the right notion for many applications. We will now consider a different notion of pseudorandomness.

A popular way to test strings for pseudorandomness before the advent of cryptography was to run “statistical tests” on them: e.g., counting the number of 0’s and 1’s, seeing if the longest run of consecutive 0’s is what you’d expect it to be in a random string, etc. As cryptographers, we want our random strings to be secure not just against *some* statistical tests, but against *any* statistical test that the adversary can devise. We therefore consider any polynomial-time algorithm that outputs 0 or 1 to be a *statistical test*.

So, let *T* be a statistical test. Then consider two experiments: `experiment-pr` and `experiment-r`. The first is as follows:

1. Select random *x* of length *k*
2. Compute $y = G(x)$
3. Run *T*(*y*) and output whatever it does

The second is as follows:

² It was similar to the next-bit-unpredictability, but allowed to choose *any* bits of *y* to disclose and then pick *any* undisclosed bit of *y* to guess.

1. Select random y of length $l(k)$
2. Run $T(y)$ and output whatever it does

Definition 4 ([Yao82]). G passes all statistical tests if for all T , there exists a negligible function $\eta(k)$ such that for all k ,

$$|\Pr[\mathbf{experiment-pr}(k) \rightarrow 1] - \Pr[\mathbf{experiment-r}(k) \rightarrow 1]| \leq \eta(k).$$

(Here and below, for ease of notation, we will often use \rightarrow instead of “outputs.”)

Note that the definition above means that a pseudorandom string can be used in place of a random one in *any* polynomial-time computation without any noticeable effect. Thus, this definition of pseudorandomness is useful also outside cryptography, for any randomized computation (e.g., Monte-Carlo simulations, primality testing, etc.).

Theorem 1 ([Yao82]³). An algorithm G is a pseudorandom generator if and only if it passes all statistical tests.

Proof. Suppose G is not a pseudorandom generator. Then let A be the predictor for G . Consider the following statistical test T : run A , get its guess for some bit, check if the guess was correct. If yes, output 1 and otherwise output 0. If A predicts with probability $1/2 + \epsilon(k)$, then $\Pr[\mathbf{experiment-pr}(k) \rightarrow 1] = 1/2 + \epsilon(k)$. Of course, there is no way to predict a truly random string, so $\Pr[\mathbf{experiment-r}(k) \rightarrow 1] = 1/2$. So the difference is $\epsilon(k)$, which is not negligible by assumption on A , so G fails the statistical test T .

The converse is a bit harder to prove. Suppose T is a statistical test that G fails. We need to build a bit-predictor A . Consider the following $l(k) + 1$ experiments, labeled $\mathbf{exp}_0, \dots, \mathbf{exp}_{l(k)}$. Experiment \mathbf{exp}_i is as follows:

1. Select random x of length k
2. Compute $y = G(x)$
3. Select a random r of length $l(k)$
4. Let z be an $l(k)$ -bit string consisting of the first i bits of y and last $l(k) - i$ bits of r
5. Run $T(z)$ and output whatever it does

Note that \mathbf{exp}_0 is exactly $\mathbf{experiment-r}$, and $\mathbf{exp}_{l(k)}$ is exactly $\mathbf{experiment-pr}$. The rest are called “hybrids.”

Suppose

$$|\Pr[\mathbf{experiment-r}(k) \rightarrow 1] - \Pr[\mathbf{experiment-pr}(k) \rightarrow 1]| \geq \epsilon(k).$$

Consider the $l(k)$ differences

$$\epsilon_i(k) \stackrel{\text{def}}{=} |\Pr[\mathbf{exp}_i(k) \rightarrow 1] - \Pr[\mathbf{exp}_{i+1}(k) \rightarrow 1]|.$$

By triangle inequality (which states that $|a - b| + |b - c| \geq |a - c|$), at least one must be greater than $\epsilon(k)/l(k)$. Indeed, suppose not. Then $\sum_{i=0}^{l(k)-1} \epsilon_i(k) < \epsilon(k)$. At the same time,

$$\begin{aligned} \sum_{i=0}^{l(k)-1} \epsilon_i(k) &= \sum_{i=0}^{l(k)-1} |\Pr[\mathbf{exp}_i(k) \rightarrow 1] - \Pr[\mathbf{exp}_{i+1}(k) \rightarrow 1]| \geq \left| \Pr[\mathbf{exp}_0 \rightarrow 1] - \Pr[\mathbf{exp}_{l(k)} \rightarrow 1] \right| = \\ &= |\Pr[\mathbf{experiment-pr}(k) \rightarrow 1] - \Pr[\mathbf{experiment-r}(k) \rightarrow 1]| \geq \epsilon(k) \end{aligned}$$

³ This theorem was attributed in [BM84] to [Yao82]. However, it did not appear in [Yao82], and moreover, it is clear that at the time of writing, Yao was not aware of it: in Section 2.3(b) of his paper, he questioned whether the Blum-Micali generator, which he knew satisfied the unpredictability definition, would pass all statistical tests. For those wondering about timing, note that conference version of [BM84] appeared in the same conference as [Yao82]. From what I understand, Yao found a proof of this theorem after his paper was published, and never published the proof. A good exposition of the proof is in [Gol01].

by triangle inequality, which is a contradiction).

Thus, there is at least one hybrid, say hybrid number j , for which

$$|\Pr[\mathbf{exp}_j(k) \rightarrow 1] - \Pr[\mathbf{exp}_{j+1}(k) \rightarrow 1]| \geq \epsilon(k)/l(k).$$

This allows us to build a bit predictor A for the bit $j + 1$.

First of all, assume that $\Pr[\mathbf{exp}_{j+1}(k) \rightarrow 1] - \Pr[\mathbf{exp}_j(k) \rightarrow 1] \geq \epsilon(k)/l(k)$, thus removing the absolute value (note that it could be the other way, with $\Pr[\mathbf{exp}_j(k) \rightarrow 1] - \Pr[\mathbf{exp}_{j+1}(k) \rightarrow 1] \geq \epsilon(k)/l(k)$, in which case our bit predictor would have to be appropriately modified; the important thing is that one of these two holds). Then $A(1^k)$ will run as follows:

1. Request the first j bits of the pseudorandom string y
2. Select a random bit g
3. Select a random r of length $l(k) - j - 1$
4. Let z be an $l(k)$ -bit string consisting of the first j bits of y followed by g followed by r
5. Run $T(z)$
6. If $T(z)$ outputs 1, output $b = g$; else output $b = 1 - g$

We have to compute the probability that b is correct. Let y_{j+1} denote the bit that A is supposed to guess (the $j + 1$ -th bit of y). Then

$$\Pr[b = y_{j+1}] = \Pr[T(z) = 1 \wedge y_{j+1} = g] + \Pr[T(z) = 0 \wedge y_{j+1} = 1 - g].$$

Now let z_1 be an $l(k)$ -bit string consisting of the first j bits of y followed by y_{j+1} followed by r , and z_2 be an $l(k)$ -bit string consisting of the first j bits of y followed by $1 - y_{j+1}$ followed by r . Note that z_1 and z_2 differ only in the bit $j + 1$; moreover, $z = z_1$ if $y_{j+1} = g$ and $z = z_2$ if $y_{j+1} = 1 - g$. Then the above probability is

$$\Pr[b = y_{j+1}] = \Pr[T(z_1) = 1 \wedge y_{j+1} = g] + \Pr[T(z_2) = 0 \wedge y_{j+1} = 1 - g].$$

Note that the events in both terms of the sum are independent now (because z_1 and z_2 don't depend on g). Note also that $\Pr[y_{j+1} = g] = \Pr[y_{j+1} = 1 - g] = 1/2$, because g is a random bit. So the probability becomes

$$\begin{aligned} \Pr[b = y_{j+1}] &= \frac{1}{2} (\Pr[T(z_1) = 1] + \Pr[T(z_2) = 0]) \\ &= \frac{1}{2} (\Pr[T(z_1) = 1] + 1 - \Pr[T(z_2) = 1]) \\ &= \frac{1}{2} + \frac{\Pr[T(z_1) = 1] - \Pr[T(z_2) = 1]}{2}. \end{aligned}$$

Now consider $\Pr[\mathbf{exp}_j(k) \rightarrow 1]$. Note that the $j + 1$ -th bit is truly random, so there are two cases: the bit could be y_{j+1} or not. So

$$\Pr[\mathbf{exp}_j(k) \rightarrow 1] = \frac{1}{2} (\Pr[T(z_1) = 1] + \Pr[T(z_2) = 1]).$$

At the same time, in experiment $j + 1$, T gets exactly z_1 . So

$$\Pr[\mathbf{exp}_{j+1}(k) \rightarrow 1] = \Pr[T(z_1) = 1].$$

Subtracting the two, we get

$$\frac{1}{2} (\Pr[T(z_1) = 1] - \Pr[T(z_2) = 1]) = \Pr[\mathbf{exp}_{j+1}(k) \rightarrow 1] - \Pr[\mathbf{exp}_j(k) \rightarrow 1] \geq \epsilon(k)/l(k),$$

so

$$\Pr[b = y_{j+1}] \geq \frac{1}{2} + \epsilon(k)/l(k).$$

Thus, if $\epsilon(k)$ is not negligible, then A can predict the $(j + 1)$ -th bit with advantage that's not negligible.

We're almost done with the proof, except that there are two good questions to ask: how does A know which bit to predict, and how does A know how to get rid of the absolute value (i.e., whether T is more likely to say 1 on experiment j or $j + 1$)⁴. For the second question, the answer is that for infinitely many values of k , at least one of the two ways of getting rid of the absolute value will work. Use that one, and then your predictor, even though incorrect for some k , will still predict with advantage that is not negligible for infinitely many k . For the first question, it turns out that simply picking i at random works, and doesn't even change the resulting success probability. For details see [Gol01].

2.4 Implications

Order Doesn't Matter What we just showed is the equivalence of next-bit-unpredictability and indistinguishability. This implies, as we prove below, that next-bit unpredictability is as good as previous-bit unpredictability.

Recall that originally the Blum-Micali generator output bits backwards. Now we know that we don't have to do that: we can run the Blum-Micali generator in the other direction, and still maintain unpredictability. This is much nicer, because we don't have to know in advance how many pseudorandom bits we will need; moreover, we need not store them all before we output them, and can do it in real time. We simply need to keep three values p, g and x_i at any given time, and we can use a single seed for as long as we want (for any polynomial number of output bits).

We now formally prove that order doesn't matter.

Theorem 2. *Let G be a pseudorandom generator. Let \tilde{G} be the algorithm that runs G and outputs its bits in reverse order. Then \tilde{G} is also a pseudorandom generator.*

Proof. We'll do this one in excruciating detail, as the first example.

Let T be a distinguisher for \tilde{G} . Consider now the following two experiments, **experiment-pr \tilde{G}** and **experiment-r \tilde{G}** : The first is as follows:

1. Select random x of length k
2. Compute $y = \tilde{G}(x)$
3. Run $T(y)$ and output whatever it does

The second is as follows:

1. Select random y of length $l(k)$
2. Run $T(y)$ and output whatever it does

We need to prove that $|\Pr[\mathbf{experiment-pr}_{\tilde{G}}(k) \rightarrow 1] - \Pr[\mathbf{experiment-r}_{\tilde{G}}(k) \rightarrow 1]|$ is negligible.

Define \tilde{T} to be the following algorithm: on input z , let \tilde{z} be z with order of bits reversed; run $T(\tilde{z})$ and output whatever it does. Now consider the following two experiments, **experiment-pr G** and **experiment-r G** . The first is as follows:

1. Select random x of length k
2. Compute $y = G(x)$

⁴ If we don't answer these questions, then we have to "hardwire" this knowledge into A , for each value of k . Then we don't quite get an algorithm, but rather an infinite family of algorithms, one for each value of k . This is called "a non-uniform algorithm" or a "circuit family." But our definition of unpredictability required an algorithm, not a circuit family.

3. Run $\tilde{T}(y)$ and output whatever it does

The second is as follows:

1. Select random y of length $l(k)$
2. Run $\tilde{T}(y)$ and output whatever it does

We know (because G is pseudorandom) that $|\Pr[\text{experiment-pr}_G(k) \rightarrow 1] - \Pr[\text{experiment-r}_G(k) \rightarrow 1]|$ is negligible. Also note that $\text{experiment-pr}_G = \tilde{T}(G(x)) = T(\tilde{G}(x)) = \text{experiment-pr}_{\tilde{G}}$ and so $\Pr[\text{experiment-pr}_G(k) \rightarrow 1] = \Pr[\text{experiment-pr}_{\tilde{G}}(k) \rightarrow 1]$. And $\Pr[\text{experiment-r}_G(k) \rightarrow 1] = \Pr[\text{experiment-r}_{\tilde{G}}(k) \rightarrow 1]$, because both run T on a uniformly distributed random string. Hence, $|\Pr[\text{experiment-pr}_{\tilde{G}}(k) \rightarrow 1] - \Pr[\text{experiment-r}_{\tilde{G}}(k) \rightarrow 1]| = |\Pr[\text{experiment-pr}_G(k) \rightarrow 1] - \Pr[\text{experiment-r}_G(k) \rightarrow 1]|$ and is negligible.

Pseudorandom is as Good as Random The notion of indistinguishability is extremely strong and has applications outside cryptography. It says that a pseudorandom string cannot be distinguished from random by *any* polynomial-time computation. Therefore, pseudorandom strings can be used in place of random ones in any feasible computation without noticeably affecting the outcome. Thus, if you have any randomized algorithm (Monte-Carlo simulation, primality test, etc.) that requires a lot of random bits, you can choose a sufficiently long seed and generate pseudorandom bits instead, provided you use a cryptographically strong pseudorandom generator as we defined it (instead of, say, the `rand` function in your software library, which usually doesn't satisfy our strong definitions, and is distinguishable by at least some polynomial-time machines).

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.
- [Gol01] Oded Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2001.
- [Yao82] A. C. Yao. Theory and application of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science*, pages 80–91, Chicago, Illinois, 3–5 November 1982. IEEE.