

Notes for Lectures 6–7

1 Indistinguishability

A popular way to test strings for pseudorandomness before the advent of cryptography was to run “statistical test” 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:

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

Definition 1 ([Yao82]). G passes all statistical tests if for all T , there exists a negligible function $\text{negl}(k)$ such that for all k ,

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

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

¹This 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].

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 $\Pr[\text{experiment-pr}(k) \rightarrow 1] = 1/2 + \epsilon(k)$. Of course, there is no way to predict a truly random string, so $\Pr[\text{experiment-pr}(k) \rightarrow 1] = 1/2$. 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 $\text{exp}_0, \dots, \text{exp}_{l(k)}$. Experiment 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 exp_0 is exactly **experiment-r**, and $\text{exp}_{l(k)}$ is exactly **experiment-pr**. The rest are called “hybrids.”

Suppose

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

Consider the $l(k)$ differences

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

By triangle inequality, 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,

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

by triangle inequality, which is a contradiction).

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

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

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

First of all, assume that $\Pr[\text{exp}_{j+1}(k) \rightarrow 1] - \Pr[\text{exp}_j(k) \rightarrow 1] \geq \epsilon(k)/l(k)$, thus removing the absolute value (note that it could be the other way, with $\Pr[\text{exp}_j(k) \rightarrow 1] - \Pr[\text{exp}_{j+1}(k) \rightarrow 1] \geq \epsilon(k)/l(k)$, 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 the 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 $\Pr[\text{exp}_j(k) \rightarrow 1]$. 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

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

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

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

Subtracting the two, we get

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

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 $j + 1$)². 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]. \square

2 Implications

2.1 Order Doesn't Matter

This implies that next-bit unpredictability is as good as previous-bit unpredictability (we prove below). Hence, 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).

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 $|\Pr[\text{experiment-pr}_{\tilde{G}}(k) \rightarrow 1] - \Pr[\text{experiment-r}_{\tilde{G}}(k) \rightarrow 1]|$ 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 $y = G(x)$
3. Run $\tilde{T}(y)$ and output whatever it does

²If 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.

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. \square

2.2 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).

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.
- [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.