CPSC 467b: Cryptography and Computer Security
Lecture 21

Michael J. Fischer

Department of Computer Science
Yale University

April 7, 2010

1. Pseudorandom Sequence Generation
   - Concepts of Pseudorandomness
   - BBS Pseudorandom Sequence Generator

Pseudorandom Sequence Generation

Pseudorandom sequence generators revisited

Cryptographically strong pseudorandom sequence generators were introduced in Lecture 6 in connection with stream ciphers.

We now show how to build one that is provably secure.

It uses the quadratic residuosity assumption (Lecture 13) on which the Goldwasser-Micali probabilistic cryptosystem is based.

Desired properties of a PRSG

A pseudorandom sequence generator (PRSG) maps a "short" random seed to a "long" pseudorandom bit string.

We want a PRSG to be *cryptographically strong*, that is, it must be difficult to correctly predict any generated bit, even knowing all of the other bits of the output sequence.

In particular, it must also be difficult to find the seed given the output sequence, since otherwise, the whole sequence is easily generated.

Thus, a PRSG is a one-way function and more.

Note: While a hash function might generate hash values of the form *yy* and still be strongly collision-free, such a function could not be a PRSG since it would be possible to predict the second half of the output knowing the first half.

Expansion amount

I am being intentionally vague about how much expansion we expect from a PRSG that maps a "short" seed to a "long" pseudorandom sequence.

Intuitively, "short" is a length like we use for cryptographic keys—long enough to prevent brute-force attacks, but generally much shorter than the data we want to deal with. Typical seed lengths might range from 128 to 2048.

By "long", we mean much larger sizes, perhaps thousands or even millions of bits, but polynomially related to the seed length.

Incremental generators

In practice, the output length is usually variable. We can request as many output bits from the generator as we like (within limits), and it will deliver them.

In this case, “long” refers to the maximum number of bits that can be delivered while still maintaining security.

Also, in practice, the bits are generally delivered a few at a time rather than all at once, so we don’t need to announce in advance how many bits we want but can go back as needed to get more.

Notation for PRSG's

In a little more detail, a *pseudorandom sequence generator* G is a function from a domain of *seeds* $\mathcal{S}$ to a domain of strings $\mathcal{X}$.

We will generally assume that all of the seeds in $\mathcal{S}$ have the same length n and that $\mathcal{X}$ is the set of all binary strings of length $\ell = \ell(n)$, where $\ell(\cdot)$ is a polynomial and $n \ll \ell(n)$.

$\ell(\cdot)$ is called the *expansion factor* of G.

What does it mean for a string to look random?

Intuitively, we want the strings $G(s)$ to "look random".
But what does it mean to "look random"?

Chaitin and Kolmogorov defined a string to be "random" if its shortest description is almost as long as the string itself.

By this definition, most strings are random by a simple counting argument.

For example, 011011011011011011011011011 is easily described as the pattern 011 repeated 9 times. On the other hand, 101110100010100101001000001 has no obvious short description.

While philosophically very interesting, these notions are somewhat different than the statistical notions that most people mean by randomness and do not seem to be useful for cryptography.

Randomness based on probability theory

We take a different tack.

We assume that the seeds are chosen truly at random from $\mathcal{S}$ according to the uniform distribution.

Let S be a uniformly distributed random variable over $\mathcal{S}$.

Then $X \in \mathcal{X}$ is a derived random variable defined by $X = G(S)$.

For $x \in \mathcal{X}$,

$$P[X = x] = \frac{|\{s \in \mathcal{S} \mid G(s) = x\}|}{|\mathcal{S}|}.$$

Thus, $P[X = x]$ is the probability of obtaining x as the output of the PRSG for a randomly chosen seed.

Randomness amplifier

We think of $G(\cdot)$ as a *randomness amplifier*.

We start with a short truly random seed and obtain a long string that "looks like" a random string, even though we know it's not uniformly distributed.

In fact, the distribution $G(S)$ is very much non-uniform.

Because $|\mathcal{S}| \leq 2^n$, $|\mathcal{X}| = 2^\ell$, and $n \ll \ell$, most strings in $\mathcal{X}$ are not in the range of G and hence have probability 0.

For the uniform distribution U over $\mathcal{X}$, all strings have the same non-zero probability $1/2^\ell$.

U is what we usually mean by a *truly random* variable on ℓ-bit strings.

Computational indistinguishability

We have already seen that the probability distributions of $X = G(S)$ and U are quite different.

Nevertheless, it may be the case that all feasible probabilistic algorithms behave essentially the same whether given a sample chosen according to X or a sample chosen according to U.

If that is the case, we say that X and U are *computationally indistinguishable* and that G is a *cryptographically strong* pseudorandom sequence generator.

Some implications of computational indistinguishability

Before going further, let me describe some functions G for which $G(S)$ is readily distinguished from U.

Suppose every string $x = G(s)$ has the form $b_1 b_1 b_2 b_2 b_3 b_3 \ldots$, for example 0011111100001100110000. . . .

Algorithm $A(x)$ outputs "G" if x is of the special form above, and it outputs "U" otherwise.

A will always guess correctly for inputs from $G(S)$, and its error probability on strings from U is only

$$\frac{2^{\ell/2}}{2^{\ell}} = \frac{1}{2^{\ell/2}}.$$

Judges

Formally, a *judge* is a probabilistic algorithm J that takes an ℓ-bit string as input and produces a single bit b as output.

Thus, it defines a *random function* from $\mathcal{X}$ to $\{0, 1\}$.

This means that for every input x, the output is 1 with some probability p_x, and the output is 0 with probability $1 - p_x$.

If the input string is a random variable X, then the probability that the output is 1 is the weighted sum of p_x over all possible inputs x, where the weight is the probability $P[X = x]$ of input x occurring.

Thus, the output value is itself a random variable $J(X)$, where

$$P[J(X) = 1] = \sum_{x \in \mathcal{X}} P[X = x] \cdot p_x.$$

Formal definition of indistinguishability

Two random variables X and Y are *ϵ-indistinguishable by judge* J if

$$|P[J(X) = 1] - P[J(Y) = 1]| < \epsilon.$$

Intuitively, we say that G is *cryptographically strong* if $G(S)$ and U are ϵ-indistinguishable for suitably small ϵ by all judges that do not run for too long.

A careful mathematical treatment of the concept of indistinguishability must relate the length parameters n and ℓ, the error parameter ϵ, and the allowed running time of the judges

Further formal details may be found in Katz and Lindell and in handout 14.

BBS Pseudorandom Sequence Generator

A cryptographically strong PRSG

We present a cryptographically strong pseudorandom sequence generator due to Blum, Blum, and Shub (BBS).

BBS is defined by a Blum integer $n = pq$ and an integer ℓ.

It maps strings in $\mathbf{Z}_n^*$ to strings in $\{0,1\}^\ell$.

Given a seed $s_0 \in \mathbf{Z}_n^*$, we define a sequence $s_1, s_2, s_3, \ldots, s_\ell$, where $s_i = s_{i-1}^2 \bmod n$ for $i = 1, \ldots, \ell$.

The ℓ-bit output sequence $\mathrm{BBS}(s_0)$ is $b_1, b_2, b_3, \ldots, b_\ell$, where $b_i = \mathrm{lsb}(s_i)$ is the least significant bit of s_i.

Recall QR assumption and Blum integers

The security of BBS is based on the assumed difficulty of determining, for a given $a \in \mathbf{Z}_n^*$ with Jacobi symbol 1, whether or not a is a quadratic residue, i.e., whether or not $a \in \mathrm{QR}_n$.

Recall from Lecture 20 that a Blum prime is a prime p such $p \equiv 3 \pmod{4}$, and a Blum integer is a number $n = pq$, where p and q are distinct Blum primes.

Also, Blum primes and Blum integers have the important property that every quadratic residue a has exactly one square root y which is itself a quadratic residue.

Call such a y the *principal square root* of a and denote it by $\sqrt{a} \pmod{n}$ or simply by $\sqrt{a}$ when it is clear that mod n is intended.

Facts about Blum integers - Jacobi symbol

Fact

Let n be a Blum integer and $a \in \mathrm{QR}_n$. *Then* $\left(\frac{a}{n}\right) = \left(\frac{-a}{n}\right) = 1$.

Proof.

This follows from the fact that if a is a quadratic residue modulo a Blum prime, then $-a$ is a quadratic non-residue. Hence,

$$\left(\frac{a}{p}\right) = -\left(\frac{-a}{p}\right) \text{ and } \left(\frac{a}{q}\right) = -\left(\frac{-a}{q}\right), \text{ so}$$

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p}\right) \cdot \left(\frac{a}{q}\right) = \left(-\left(\frac{-a}{p}\right)\right) \cdot \left(-\left(\frac{-a}{q}\right)\right) = \left(\frac{-a}{n}\right).$$

$\square$

Facts about Blum integers - lsb

The second fact simply says that the low-order bits of $x \bmod n$ and $(-x) \bmod n$ always differ when n is odd.

Let $\operatorname{lsb}(x) = (x \bmod 2)$ be the least significant bit of integer x.

Fact

Let n be odd. Then $\operatorname{lsb}(x \bmod n) \oplus \operatorname{lsb}((-x) \bmod n) = 1$.

First-bit prediction

A *first-bit predictor with advantage* ϵ is a probabilistic polynomial time algorithm A that, given $b_2, \ldots, b_\ell$, correctly predicts b_1 with probability at least $1/2 + \epsilon$.

This is not sufficient to establish that the pseudorandom sequence $\mathrm{BBS}(S)$ is indistinguishable from the uniform random sequence U, but if it did not hold, there certainly would exist a distinguishing judge.

Namely, the judge that outputs 1 if $b_1 = A(b_2, \ldots, b_\ell)$ and 0 otherwise would output 1 with probability greater than $1/2 + \epsilon$ in the case that the sequence came from $\mathrm{BBS}(S)$ and would output 1 with probability exactly $1/2$ in the case that the sequence was truly random.

BBS has no first-bit predictor under the QR assumption

If BBS has a first-bit predictor A with advantage ϵ, then there is a probabilistic polynomial time algorithm Q for testing quadratic residuosity with the same accuracy.

Thus, if quadratic-residue-testing is “hard”, then so is first-bit prediction for BBS.

Theorem

Let A be a first-bit predictor for BBS(S) with advantage ϵ. Then we can find an algorithm Q for testing whether a number x with Jacobi symbol 1 is a quadratic residue, and Q will be correct with probability at least $1/2 + \epsilon$.

Construction of Q

Assume that A predicts b_1 given $b_2, \ldots, b_\ell$.

Algorithm $Q(x)$ tests whether or not a number x with Jacobi symbol 1 is a quadratic residue modulo n.

It outputs 1 to mean $x \in \mathrm{QR}_n$ and 0 to mean $x \notin \mathrm{QR}_n$.

```
To Q(x):
1.  Let $\hat{s}_2 = x^2 \bmod n$.
2.  Let $\hat{s}_i = \hat{s}_{i-1}^2 \bmod n$, for $i = 3, \ldots, \ell$.
3.  Let $\hat{b}_1 = \mathrm{lsb}(x)$.
4.  Let $\hat{b}_i = \mathrm{lsb}(\hat{s}_i)$, for $i = 2, \ldots, \ell$.
5.  Let $c = A(\hat{b}_2, \ldots, \hat{b}_\ell)$.
6.  If $c = \hat{b}_1$ then output 1; else output 0.
```

Why Q works

Since $\left(\frac{x}{n}\right) = 1$, then either x or $-x$ is a quadratic residue. Let s_0 be the principal square root of x or $-x$. Let $s_1, \ldots, s_\ell$ be the state sequence and $b_1, \ldots, b_\ell$ the corresponding output bits of BBS(s_0).

We have two cases.

Case 1: $x \in \mathrm{QR}_n$. Then $s_1 = x$, so the state sequence of BBS(s_0) is

$$s_1, s_2, \ldots, s_\ell = x, \hat{s}_2, \ldots, \hat{s}_\ell,$$

and the corresponding output sequence is

$$b_1, b_2, \ldots, b_\ell = \hat{b}_1, \hat{b}_2, \ldots, \hat{b}_\ell.$$

Since $\hat{b}_1 = b_1$, $Q(x)$ correctly outputs 1 whenever A correctly predicts b_1. This happens with probability at least $1/2 + \epsilon$.

Why Q works (cont.)

Case 2: $x \in \mathrm{QNR}_n$, so $-x \in \mathrm{QR}_n$. Then $s_1 = -x$, so the state sequence of BBS(s_0) is

$$s_1, s_2, \ldots, s_\ell = -x, \hat{s}_2, \ldots, \hat{s}_\ell,$$

and the corresponding output sequence is

$$b_1, b_2, \ldots, b_\ell = \neg\hat{b}_1, \hat{b}_2, \ldots, \hat{b}_\ell.$$

Since $\hat{b}_1 = \neg b_1$, $Q(x)$ correctly outputs 0 whenever A correctly predicts b_1. This happens with probability at least $1/2 + \epsilon$.
In both cases, $Q(x)$ gives the correct output with probability at least $1/2 + \epsilon$.