CPSC 467b: Cryptography and Computer Security

Michael J. Fischer

Lecture 18
March 26, 2012

Pseudorandom Sequence Generation

BBS Pseudorandom Sequence Generator

Bit-Prediction

Pseudorandom Sequence Generation

Pseudorandom sequence generators revisited

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

We next define carefully what it means for a pseudorandom sequence generator (PRSG) to be *cryptographically strong*.

We then show how to build one that is provably secure. It is based on the quadratic residuosity assumption (Lecture 14) 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 uniformly at random from $\mathcal{S}$.

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 $x \in \mathcal{X}$,

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

Thus, $\Pr[\mathrm{X} = \mathrm{x}]$ 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 just 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 00111111000011001100000....

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

A will always output "G" for inputs from $G(S)$. For inputs from U, A will output "G" with probability only

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

How many strings of length ℓ have the special form above?

Judges

Formally, a *judge* is a probabilistic polynomial-time algorithm J that takes an ℓ-bit input string x and outputs a single bit b.

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 $\Pr[X = x]$ of input x occurring.

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

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

Formal definition of indistinguishability

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

$$|\Pr[\mathrm{J}(\mathrm{X}) = 1] - \Pr[\mathrm{J}(\mathrm{Y}) = 1]| < \epsilon.$$

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 Goldwasser and Bellare and in handout 9.

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 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.

Security of BBS

In the next several slides, we show that BBS is secure.

The proof reduces the problem of predicting the output of BBS to the quadratic residue problem.

We finally show that if there is a judge that successfully distinguishes BBS(S) from U, then there is a feasible method for distinguishing quadratic residues from non-residues with Jacobi symbol 1.

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 17 that a Blum prime is a prime $p \equiv 3 \pmod 4$, 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.

Blum integers and the 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,

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

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

□

Blum integers and the least significant bit

The low-order bits of x mod n and $(-x)$ mod n always differ when n is odd.

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

Fact

If n is odd, then $\mathrm{lsb}(x \bmod n) \oplus \mathrm{lsb}((-x) \bmod n) = 1$.

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 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 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 $1/2 + \epsilon$.

Bit-Prediction

Bit-prediction and statistical independence

One important property of the uniform distribution U on bit-strings $b_1, \ldots, b_\ell$ is that the individual bits are statistically independent from each other.

This means that the probability that a particular bit $b_i = 1$ is unaffected by the values of the other bits in the sequence.

Thus, any algorithm that attempts to predict b_i, even knowing other bits of the sequence, will be correct only $1/2$ of the time.

We now translate this property of unpredictability to pseudorandom sequences.

Next-bit prediction

First-bit prediction seems rather uninteresting because pseudorandom bits are usually generated in order.

However, we would like it to be difficult to predict the next bit given the bits that came before.

Algorithm A is an *ϵ-next-bit predictor for bit i* if

$$\Pr[\mathrm{A}(\mathrm{b}_1, \ldots, \mathrm{b}_{\mathrm{i}-1}) = \mathrm{b}_\mathrm{i}] \geq \frac{1}{2} + \epsilon$$

where $(b_1, \ldots, b_i) = G_i(S)$.

As before, S is uniformly distributed over $\mathcal{S}$, $G(S)$ is a random variable over the output strings of G, and $G_i(S)$ is the corresponding random variable on the length-i prefixes of $G(S)$.

Next-bit prediction and indistinguishability

Next-bit prediction is closely related to indistinguishability.

Roughly speaking, $G(S)$ has a next-bit predictor for some bit i iff $G(S)$ is distinguishable from U.

The precise definitions under which this theorem is true are subtle, for one must quantify both the amount of time the judge and next-bit predictor algorithms are permitted to run as well as how much better than chance the judgments or predictions must be in order to be considered a successful judge or next-bit predictor.

We defer the mathematics for now and focus instead on the intuitive concepts that underlie this theorem.

Building a judge from a next-bit predictor

Let A be an ϵ-next-bit predictor for G for some bit i.

Here's how to build a judge J that distinguishes $G(S)$ from U with advantage ϵ.

- J, given a sample x drawn from either $G(S)$ or from U, runs $A(x)$ to produce $\hat{b}_i$.
- If $\hat{b}_i = b_i$, then J outputs 1.
- Otherwise, J outputs 0.

The advantage of J

For samples from $G(S)$, the judge will output 1 with the same probability that A successfully predicts bit b_i, which is at least $1/2 + \epsilon$.

For sequences drawn from U, the judge will output 1 with probability exactly $1/2$.

Hence, the judge distinguishes $G(S)$ from U with advantage ϵ.

It follows that no cryptographically strong PRSG can have an ϵ-next-bit predictor.

In other words, no algorithm that attempts to predict the next bit can have more than a "small" advantage ϵ over chance.

Previous-bit prediction

Previous-bit prediction, while perhaps less natural, is analogous to next-bit prediction.

An *ϵ-previous-bit predictor for bit i* is a probabilistic polynomial time algorithm A that, given bits $b_{i+1}, \ldots, b_\ell$, correctly predicts b_i with probability at least $1/2 + \epsilon$.

As with next-bit predictors, if $G(S)$ has a previous-bit predictor for some bit b_j, then some judge distinguishes $G(S)$ from U.

Again, I am being vague with the exact conditions under which this is true.

Hence, $G(S)$ has an ϵ-next-bit predictor for some bit i if and only if it has an ϵ'-previous-bit predictor for some bit j (where ϵ and ϵ' are related but not necessarily equal).

Special case of $\ell = 2$

To give some intuition into why such a fact might be true, we look at the special case of $\ell = 2$, that is, of 2-bit sequences.

The probability distribution $G(S)$ can be described by four probabilities

$$p_{u,v} = \Pr[\mathrm{b}_1 = \mathrm{u} \wedge \mathrm{b}_2 = \mathrm{v}], \text{ where } \mathrm{u}, \mathrm{v} \in \{0, 1\}.$$

Written in tabular form, we have

		b_2 = 0	b_2 = 1
b_1	0	$p_{0,0}$	$p_{0,1}$
	1	$p_{1,0}$	$p_{1,1}$

Bit prediction when $\ell = 2$

We describe a deterministic algorithm $A(v)$ for predicting b_1 given $b_2 = v$. $A(v)$ predicts $b_1 = 0$ if $p_{0,v} > p_{1,v}$, and it predicts $b_1 = 1$ if $p_{0,v} \leq p_{1,v}$.

In other words, the algorithm chooses the value for b_1 that is most likely given that $b_2 = v$.

Theorem

If A is an ϵ-previous-bit predictor for b_1, then A is an ϵ-next-bit predictor for either b_1 or b_2.

Proof that A is a next-bit predictor

Assume A is an ϵ-previous-bit predictor for b_1, so A correctly predicts b_1 given b_2 with probability $\geq 1/2 + \epsilon$.

We show that A is an ϵ-next-bit predictor for either b_1 or b_2.

Let $a(v)$ be the value predicted by $A(v)$ for $v \in \{0, 1\}$.

We have two cases:

Case 1: $a(0) = a(1)$. Then algorithm A does not depend on v, that is, it makes the same prediction regardless of the value of v.

Thus, $A(0)$ correctly predicts b_1 with probability at least $1/2 + \epsilon$. (This means that $\Pr[\mathrm{b_1 = A(0)}] \geq 1/2 + \epsilon$.)

It follows that A is an ϵ-next-bit predictor for b_1.

Proof that A is a next-bit predictor (cont.)

Case 2: $a(0) \neq a(1)$. The probability that $A(v)$ correctly predicts b_1 given $b_2 = v$ is

$$\Pr[\mathrm{b}_1 = \mathrm{a}(\mathrm{v}) \mid \mathrm{b}_2 = \mathrm{v}] = \frac{\Pr[\mathrm{b}_1 = \mathrm{a}(\mathrm{v}) \wedge \mathrm{b}_2 = \mathrm{v}]}{\Pr[\mathrm{b}_2 = \mathrm{v}]} = \frac{\mathrm{p}_{\mathrm{a}(\mathrm{v}),\mathrm{v}}}{\Pr[\mathrm{b}_2 = \mathrm{v}]}$$

The overall probability that $A(b_2)$ is correct for b_1 is the average of the conditional probabilities for $v = 0$ and $v = 1$, weighted by the probability that $b_2 = v$. Thus,

$$\begin{aligned}
&\Pr[A(b_2) \text{ is correct for } b_1] \\
&= \sum_{v \in \{0,1\}} \Pr[\mathrm{b}_1 = \mathrm{a}(\mathrm{v}) \mid \mathrm{b}_2 = \mathrm{v}] \cdot \Pr[\mathrm{b}_2 = \mathrm{v}] \\
&= \sum_{v \in \{0,1\}} p_{a(v),v} = p_{a(0),0} + p_{a(1),1}
\end{aligned}$$

Proof that A is a next-bit predictor (cont.)

Similarly, using A to predict b_2 given b_1 yields

$$\begin{aligned}
&\Pr[A(b_1) \text{ is correct for } b_2] \\
&= \sum_{v \in \{0,1\}} \Pr[\mathrm{b}_2 = \mathrm{a}(\mathrm{u}) \mid \mathrm{b}_1 = \mathrm{u}] \cdot \Pr[\mathrm{b}_1 = \mathrm{u}] \\
&= \sum_{v \in \{0,1\}} p_{u,a(u)} = p_{0,a(0)} + p_{1,a(1)}
\end{aligned}$$

We show that

$$p_{a(0),0} + p_{a(1),1} = p_{0,a(0)} + p_{1,a(1)}$$

when $a(0) \neq a(1)$. It follows that

$$\Pr[A(b_1) \text{ is correct for } b_2] = \Pr[A(b_2) \text{ is correct for } b_1],$$

so A is an ϵ-next-bit predictor for b_2.

Proof that A is a next-bit predictor (cont.)

Since $a(0) \neq a(1)$, the function $a(\cdot)$ is one-to-one and onto, so either $a(v) = v$ for $v \in \{0, 1\}$, or $a(v) = \neg v$ for $v \in \{0, 1\}$.

That is, $a(\cdot)$ is either the identity or the complement function. Hence, either

$$p_{a(0),0} + p_{a(1),1} = p_{0,0} + p_{1,1} = p_{0,a(0)} + p_{1,a(1)}$$

or

$$p_{a(0),0} + p_{a(1),1} = p_{1,0} + p_{0,1} = p_{0,a(0)} + p_{1,a(1)}$$

as desired. Hence, A is an ϵ-next-bit predictor for b_2.

Combining the two cases, we conclude that A is an ϵ-next-bit predictor for either b_1 or b_2, proving the theorem.

Summary of results

We have just seen how to construct a next-bit predictor from a previous-bit predictor, and we've also seen how to construct a judge from a next-bit predictor.

The most general bit-prediction problem is to predict the i^{th} bit of the sequence given *all* other bits. An algorithm that can do this with advantage ϵ is said to be an *ϵ-i^{th}-bit predictor for G*.

It's easy to transform an ϵ-next-bit predictor for b_i into an ϵ-i^{th}-bit predictor.

It's also easy to build a judge with advantage ϵ from an ϵ-i^{th}-bit predictor.

To close the loop, one can build an ϵ'-next-bit predictor for some bit i and some ϵ' given a judge with advantage ϵ.

Bit-prediction given a judge

We sketch how to build a next-bit predictor given a judge.
The construction is based on interpolation between U and $G(S)$.

$$
\begin{array}{cccccccccc}
u_1 & u_2 & u_3 & \dots & u_{i-1} & u_i & u_{i+1} & \dots & u_\ell \\
b_1 & u_2 & u_3 & \dots & u_{i-1} & u_i & u_{i+1} & \dots & u_\ell \\
 & & & \dots & & & & \dots & \\
b_1 & b_2 & b_3 & \dots & b_{i-1} & u_i & u_{i+1} & \dots & u_\ell \\
b_1 & b_2 & b_3 & \dots & b_{i-1} & b_i & u_{i+1} & \dots & u_\ell \\
 & & & \dots & & & & \dots & \\
b_1 & b_2 & b_3 & \dots & b_{i-1} & b_i & b_{i+1} & \dots & b_\ell
\end{array}
$$

The difference in the judge's output between top and bottom sequence is $\geq \epsilon$.

Therefore, for some i, the difference in judge's output between sequence $i-1$ and i must be at least $\epsilon' = \epsilon/\ell$.

An ϵ'-next bit predictor for b_i is easily constructed.