CPSC 467: Cryptography and Computer Security

Michael J. Fischer

Lecture 21
November 15, 2017

Secure Random Sequence Generators
- Pseudorandom sequence generators
- Looking random

Similarity of Probability Distributions
- Cryptographically secure PRSG
- Indistinguishability

The Legendre and Jacobi Symbols
- The Legendre symbol
- Jacobi symbol
- Computing the Jacobi symbol

Secure Random Sequence Generators

Pseudorandom sequence generators

A *pseudorandom sequence generator (PRSG)* is a function that maps a short *seed* to a long "random-looking" *output sequence*.

The seed typically has length between 32 and a few thousand bits.

The output is typically much longer, ranging from thousands or millions of bits or more, but polynomially related to the seed length.

The output of a PRSG is a sequence that is supposed to "look random".

Incremental generators

In practice, a PRSG is implemented as a co-routine that outputs the next block of bits in the sequence each time it is called. For example, the linux function

```
void srandom(unsigned int seed)
```

sets the 32-bit seed. Each subsequent call on

```
long int random(void)
```

returns an integer in the range $[0, \ldots, \mathit{RAND\_MAX}]$.

On my machine, the return value is 31 bits long (even though `sizeof(long int)` is 64).

Limits on incremental generators

Incremental generators typically are based on state machines with a finite number of states, so the output eventually becomes periodic.

The period of `random()` is said to be approximately $16 * (2^{31} - 1)$.

The output of a PRSG becomes predictable from past outputs once the generator starts to repeat. The point of repetition defines the *maximum usable output length*, even if the implementation allows bits to continue to be produced.

What does it mean for a string to look random?

For the output of a PRSG to look random:

- It must pass common statistical tests of randomness. For example, the frequencies of 0's and 1's in the output sequence must be approximately equal.
- It must lack obvious structure, such as having all 1's occur in pairs.
- It must be difficult to find the seed given the output sequence, since otherwise the whole sequence is easily generated.
- It must be difficult to correctly predict any generated bit, even knowing all of the other bits of the output sequence.
- It must be difficult to distinguish its output from truly random bits.

Chaitin/Kolmogorov randomness

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.

Cryptographically secure PRSG

A PRSG is said to be *cryptographically secure* if its output cannot be *feasibly* distinguished from truly random bits.

In other words, no feasible probabilistic algorithm behaves significantly differently when presented with an output from the PRSG as it does when presented with a truly random string of the same length.

We argue that this definition encompasses all of the desired properties for "looking random" discussed earlier,

Looking ahead

In the rest of this lecture, we carefully define what it means for a PRSG to be secure.

We then show how to build a PRSG that is provably secure. It is based on the *quadratic residuosity assumption* (lecture 20) on which the Goldwasser-Micali probabilistic cryptosystem is based.

Similarity of Probability Distributions

Formal definition of PRSG

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

We 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 $\ell = \ell(n)$.

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

$\ell(\cdot)$ is assumed to be a polynomial such that $n \ll \ell(n)$.

Output distribution of a PRSG

Let S be a uniformly distributed random variable over the set $\mathcal{S}$ of possible seeds.

The *output distribution* of G is a random variable $X \in \mathcal{X}$ defined by $X = G(S)$.

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

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

Thus, $\Pr[X = x]$ 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 random string distributed according to X, which 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 secure* 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 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 *probabilistic 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 $\mathrm{Pr}[X = x]$ of input x occurring.

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

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

Formal definition of indistinguishability

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

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

Intuitively, we say that G is *cryptographically secure* 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.

The Legendre and Jacobi Symbols

Notation for quadratic residues

The Legendre and Jacobi symbols form a kind of calculus for reasoning about quadratic residues and non-residues.

They lead to a feasible algorithm for determining membership in $Q_n^{01} \cup Q_n^{10}$. Like the Euclidean gcd algorithm, the algorithm does not require factorization of its arguments.

The existence of this algorithm also explains why the Goldwasser-Micali cryptosystem can't use all of QNR_n in the encryption of "1", for those elements in $Q_n^{01} \cup Q_n^{10}$ are readily determined to be in QNR_n.

Legendre symbol

Let p be an odd prime, a an integer. The *Legendre symbol* $\left(\frac{a}{p}\right)$ is a number in $\{-1, 0, +1\}$, defined as follows:

$$\left(\frac{a}{p}\right) = \begin{cases} +1 & \text{if } a \text{ is a non-trivial quadratic residue modulo } p \\ 0 & \text{if } a \equiv 0 \pmod{p} \\ -1 & \text{if } a \text{ is } \textit{not} \text{ a quadratic residue modulo } p \end{cases}$$

By the Euler Criterion, we have

Theorem

Let p be an odd prime. Then

$$\left(\frac{a}{p}\right) \equiv a^{\left(\frac{p-1}{2}\right)} \pmod{p}$$

Note that this theorem holds even when $p \mid a$.

Properties of the Legendre symbol

The Legendre symbol satisfies the following *multiplicative property*:

Fact

Let p be an odd prime. Then

$$\left(\frac{a_1 a_2}{p}\right) = \left(\frac{a_1}{p}\right)\left(\frac{a_2}{p}\right)$$

Not surprisingly, if a_1 and a_2 are both non-trivial quadratic residues, then so is $a_1 a_2$. Hence, the fact holds when

$$\left(\frac{a_1}{p}\right) = \left(\frac{a_2}{p}\right) = 1.$$

Product of two non-residues

Suppose $a_1 \notin \mathrm{QR}_p$, $a_2 \notin \mathrm{QR}_p$. The above fact asserts that the product $a_1 a_2$ *is* a quadratic residue since

$$\left(\frac{a_1 a_2}{p}\right) = \left(\frac{a_1}{p}\right)\left(\frac{a_2}{p}\right) = (-1)(-1) = 1.$$

Here's why.

- Let g be a primitive root of p.
- Write $a_1 \equiv g^{k_1} \pmod{p}$ and $a_2 \equiv g^{k_2} \pmod{p}$.
- Both k_1 and k_2 are odd since $a_1, a_2 \notin \mathrm{QR}_p$.
- But then $k_1 + k_2$ is even.
- Hence, $g^{(k_1+k_2)/2}$ is a square root of $a_1 a_2 \equiv g^{k_1+k_2} \pmod{p}$, so $a_1 a_2$ is a quadratic residue.

The Jacobi symbol

The *Jacobi symbol* extends the Legendre symbol to the case where the "denominator" is an arbitrary odd positive number n.

Let n be an odd positive integer with prime factorization $\prod_{i=1}^{k} {p_i}^{e_i}$. We define the *Jacobi symbol* by

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{k} \left(\frac{a}{p_i}\right)^{e_i} \tag{1}$$

The symbol on the left is the Jacobi symbol, and the symbol on the right is the Legendre symbol.

(By convention, this product is 1 when $k = 0$, so $\left(\frac{a}{1}\right) = 1$.)

The Jacobi symbol extends the Legendre symbol since the two definitions coincide when n is an odd prime.

Meaning of Jacobi symbol

What does the Jacobi symbol mean when n is not prime?

- If $\left(\frac{a}{n}\right) = +1$, a might or might not be a quadratic residue.
- If $\left(\frac{a}{n}\right) = 0$, then $\gcd(a, n) \neq 1$.
- If $\left(\frac{a}{n}\right) = -1$ then a is definitely not a quadratic residue.

Jacobi symbol $= +1$ for $n = pq$

Let $n = pq$ for p, q distinct odd primes. Since

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p}\right)\left(\frac{a}{q}\right) \tag{2}$$

there are two cases that result in $\left(\frac{a}{n}\right) = 1$:

1. $\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = +1$, or
2. $\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = -1$.

Case of both Jacobi symbols $= +1$

If $\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = +1$, then $a \in \mathrm{QR}_p \cap \mathrm{QR}_q = Q_n^{11}$.

It follows by the Chinese Remainder Theorem that $a \in \mathrm{QR}_n$.

This fact was implicitly used in the proof sketch that $|\sqrt{a}| = 4$.

Case of both Jacobi symbols $= -1$

If $\left(\frac{a}{p}\right) = \left(\frac{a}{q}\right) = -1$, then $a \in \mathrm{QNR}_p \cap \mathrm{QNR}_q = Q_n^{00}$.

In this case, a is *not* a quadratic residue modulo n.

Such numbers a are sometimes called "pseudo-squares" since they have Jacobi symbol 1 but are not quadratic residues.

Computing the Jacobi symbol

The Jacobi symbol $\left(\frac{a}{n}\right)$ is easily computed from its definition (equation 1) and the Euler Criterion, given the factorization of n.

Similarly, $\gcd(u, v)$ is easily computed without resort to the Euclidean algorithm given the factorizations of u and v.

The remarkable fact about the Euclidean algorithm is that it lets us compute $\gcd(u, v)$ efficiently, without knowing the factors of u and v.

A similar algorithm allows us to compute the Jacobi symbol $\left(\frac{a}{n}\right)$ efficiently, without knowing the factorization of a or n.

Identities involving the Jacobi symbol

The algorithm is based on identities satisfied by the Jacobi symbol:

1. $\left(\frac{0}{n}\right) = \begin{cases} 1 & \text{if } n = 1 \\ 0 & \text{if } n \neq 1; \end{cases}$
2. $\left(\frac{2}{n}\right) = \begin{cases} 1 & \text{if } n \equiv \pm 1 \pmod 8 \\ -1 & \text{if } n \equiv \pm 3 \pmod 8; \end{cases}$
3. $\left(\frac{a_1}{n}\right) = \left(\frac{a_2}{n}\right)$ if $a_1 \equiv a_2 \pmod n$;
4. $\left(\frac{2a}{n}\right) = \left(\frac{2}{n}\right) \cdot \left(\frac{a}{n}\right)$;
5. $\left(\frac{a}{n}\right) = \begin{cases} \left(\frac{n}{a}\right) & \text{if } a, n \text{ odd and } \neg(a \equiv n \equiv 3 \pmod 4) \\ -\left(\frac{n}{a}\right) & \text{if } a, n \text{ odd and } a \equiv n \equiv 3 \pmod 4. \end{cases}$

A recursive algorithm for computing Jacobi symbol

```
/* Precondition: a, n >= 0; n is odd */
int jacobi(int a, int n) {
  if (a == 0)                       /* identity 1 */
    return (n==1) ? 1 : 0;
  if (a == 2)                       /* identity 2 */
    switch (n%8) {
    case 1: case 7: return 1;
    case 3: case 5: return -1;
    }
  if ( a >= n )                     /* identity 3 */
    return jacobi(a%n, n);
  if (a%2 == 0)                     /* identity 4 */
    return jacobi(2,n)*jacobi(a/2, n);
  /* a is odd */                    /* identity 5 */
  return (a%4 == 3 && n%4 == 3) ? -jacobi(n,a) : jacobi(n,a);
}
```