Random and pseudo-random bit generation

Cryptography, fall 2005

Liliana Ewa Skolimowska
Jörn Martin Hajek

Introduction

In cryptography, we are often in a situation where we need a random integer or a random sequence of bits. For example, secret keys are often required to be random. But what does that really mean? If we flip a coin eight times and assign 0 to one of the sides and 1 to the other, then the sequence 00000000 is just as likely as the sequence 10011100. Still, our intuition tells us that the second sequence is "more random" than the first one. But actually, humans are quite bad at consciously creating random sequences – so should we give any credit to their intuition?

In the following pages, we will try to describe how to deal with the need for randomness – and to help people to understand what randomness is. Also, we will present our own attempts at generating random bits through human input.

Random bits

A sequence of random bits is a sequence where the bits are independent from each other and uniformly distributed.
Random bits can be generated in may different ways. As it turns out, most of these ways are rather slow and/or expensive. Random-bit generators can be based on hardware or on software. For hardware, you could use things like the time elapsed between emission of particles during radioactive decay, sounds from different sources, air turbulence within a disk drive causing random fluctuations in disk drive sector read latency times or any other phenomena that can be translated into bits – even lava lamps have been used. For software, you could use time, some kind of user input (though having a human type 1's and 0's will most likely not yield a good result,) or the system environment.
Most likely, none of these methods will give us a uniform distribution of 1's and 0's, but it can be solved by simple de-skewing techniques. For example, you can look at pairs of bits, throw them away if both bits are the same, and keep the first bit if they are different. Of course, this means you have to produce at least 4 times as many bits as you will need.
Combining any number of these methods (as long as they are independent) should yield results that are as least as good as the best of the methods. There are different ways to accomplish this, a (possibly too) simple method is to just XOR the results. A better method would be to use some kind of hash-function on a concatenation of the results.

We will present some simple examples of random number generators later – mostly to show that it's really not that trivial...

Pseudo-random bits

A way to save a lot of time is to use a pseudo-random bit generator (PRBG). This is an algorithm that takes a relatively small number of random bits (length k) as input, and outputs a longer sequence (length l) of bits, that appears to be random. These algorithms are deterministic, that means that they will always produce the same output for a given input. The intent is to make the outputs indistinguishable from truly random bit-strings.

There are many different PRBG's out there. One often used is the linear congruential generator. It produces a pseudorandom sequence of numbers through the formula

$$x_n = ax_{n-1} + b \bmod m, \quad n \geq 1;$$

where a, b and m are chosen parameters.

This is a very useful function for things like simulations or other algorithms where the randomness is more important than the secrecy, but it has been proven completely insecure for cryptographic purposes.

Secure PRBG's should must have at least the following properties:

1) The length k of the *seed* (the input to the function) should be large enough, so that an adversary cannot simply run through all possibilities

2) The output sequences should be statistically indistinguishable from random sequences (more on that later)

3) The output bits should be unpredictable to an adversary with limited resources

More specifically for 3):

The next-bit test:

A PRBG is said to pass the next-bit test if there is no polynomial-time algorithm that given bits 0 to m-1 of an output sequence can predict bit m with probability significantly higher than 50%.

A PRBG passing the next-bit test is called cryptographically secure (CSPRBG.) We will present a well-known CSPRBG later.

For 2): A random sequence is assumed to have certain properties. These properties should appear in pseudo-random sequence as well. Some of these are mentioned in Golomb's randomness postulates. For example, the number of 1's and 0's should be about the same, the bits should not depend on each other, and so on. There are tests for each of these properties. Failing one of these tests indicates non-randomness. Of course, passing one of these tests does not guarantee randomness – in fact, even passing all of these tests does not guarantee randomness. Each test just makes it more and more unlikely that a sequence is non-random.

We will now present five tests that are often used in testing PRBG's. They all test the hypothesis that the sequence was generated by a random number generator, with a significance level of 5%. That means, that sequences actually produced with a random number generator are rejected with a probability of 5%. This may seem like a lot, but it also makes it more unlikely that we will accept sequences that do not possess the randomness-properties.

The Frequency test

The purpose of this test is to see if there are about as many 1's as 0's.

n: the number of bits, n0: the number of 0's, n1: the number of 1's.

$$X_1 = \frac{(n_0 - n_1)^2}{n}$$

X_1 should be less than 3.8415

The Serial test

This test is similar to the frequency test, but looks for 2-bit subsequences instead.

n: the number of bits, n0: the number of 0's, n1: the number of 1's, n00: the number of 00's, ...

For example, for the sequence 011100, the values would be n=6, n0=3, n1=3, n00=1, n01=1, n10=1, n11=2.

$$X_2 = \frac{4}{n-1}\left(n_{00}^2 + n_{01}^2 + n_{10}^2 + n_{11}^2\right) - \frac{2}{n}\left(n_0^2 + n_1^2\right) + 1$$

X_2 should be less than 5.9915

The Poker test

We divide the sequence into subsequences of a certain length, and then test if these sequences appear the same number of times. m is a positive integer such as: $\left\lfloor \frac{n}{m} \right\rfloor \geq 5 \cdot (2^m)$, $k = \left\lfloor \frac{n}{m} \right\rfloor$, n_i is the number of occurrences of the i^{th} type of sequence of length m and $1 \leq i \leq 2^m$

The poker test is a generalisation of the frequency test: m=1 in the poker test yields the frequency test.

The statistic used is:

$$X_3 = \frac{2^m}{k}\left(\sum_{i=1}^{2^m} n_i^2\right) - k$$

X_3 should be less than 14,0671

The Runs test

A run is a subsequence , which consist of the consecutive 0's or consecutive 1's. A run of 0's is called a gap, and a run of 1's is called a block.

In runs test we check if the number of runs is as expected for a random sequence. The expected number of gaps or blocks of length i in a random sequence of length n is :

$e_i = (n-i+3)/2^{i+2}$, k is equal to the largest integer i for which $e_i \geq 5$, B_i, G_i are the number of gaps and blocks of length i in s for each i, and $1 \leq i \leq k$
The statistic used is:

$$X_4 = \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i}$$

X_4 should be less than 9,4877

The autocorrelation test

The purpose of the autocorrelation test is to check that bits don't depend on other bits. We choose a number d, and then compare each bit i to the bit i+d. If the two bits are the same too often, or different too often, the test rejects the input.

d is fixed integer such that:

$1 \leq d \leq \lfloor n/2 \rfloor$, $A(d) = \sum_{i=0}^{n-d-1} s_i \oplus s_{i+d}$

where $\oplus$ denotes the XOR operator.

$$X_5 = 2\left(A(d) - \frac{n-d}{2}\right)/\sqrt{n-d}$$

X_5 should be less than 1,96 and bigger than -1,96.

We will use these five tests later on the sequences we produced with our own programs.

One Algorithm that produces cryptographically secure pseudo-random bits is the following:

Algorithm Blum-Blum-Shub pseudorandom bit generator
SUMMARY: a pseudorandom bit sequence $z_1, z_2, \ldots, z_l$ of length l is generated.
1. *Setup*. Generate two large secret random (and distinct) primes p and q (cf. Note 8.8), each congruent to 3 modulo 4, and compute n = pq.
2. Select a random integer s (the *seed*) in the interval [1, n−1] such that gcd(s, n) = 1, and compute $x_0 \leftarrow s^2 \bmod n$.
3. For i from 1 to l do the following:

3.1 $x_i \leftarrow x^2_{i-1} \bmod n$.
3.2 $z_i \leftarrow$ the least significant bit of x_i.
4. The output sequence is $z_1, z_2, \ldots, z_l$.

Our random bit generators

As we wanted to do a bit of programming for this assignment, we decided to come up with some ideas for user-input based random bit generation, and then to test and see if our ideas were any good, and maybe combine them to make an even better generator. We started with three basic ideas:

Have the user randomly hack away on the keyboard ("typing")
Have the user randomly click on the screen ("mouse")
Measure the time it takes the user to hit a key ("time")

First, we programmed the five basic tests described above. Then we implemented our three ideas, and tested them. The mouse-click program turned out to be too simple, and we changed it. Then we combined the mouse-click and the timing programs, and finally we combined that program with the typing program. All in all, we had six small programs that we implemented, used and tested. Due to time constraints, we only made ten sequences with each program, with around 160 bits per sequence. 160 bits should be enough as seed for at least some of the PRBG's. However, both the number of sequences and the number of bits per sequence are too low to be statistically significant – we tried to interpret them the best we could anyway. Also, most of the programs could have been improved given more time – we will mention some of the ideas while describing the programs.

The programs

Our first program, "typing", is rather simple: We assign either 1 or 0 to each key on the keyboard. Ideally, we then open a text editor and get a monkey to hit the keyboard. In practice, we were forced to use computer science students instead. After producing a file like this, our program would read the file, convert the characters to the bits we assigned to the keys, and use the simple de-skewing technique we described earlier to produce our output.

There are some issues with the input. The user would often use more than one finger at the same time. While each key should be equally likely at any given point, this was not the case when more fingers where used, as keys are more unlikely to be pressed twice in succession then they should be. However, as each bit was assigned to more than 20 keys, this dependence hopefully does not show too strongly. The user may also be tempted to just just hit some keys with the left hand, then with the right hand, and then hitting the same keys with the left hand as he did the first time, etc. This was only solved by instructing the subjects too avoid it – a more complex program could have checked for behaviour like this and ignored bits produced this way.

The first try of our second program, "mouse," opens a window with four buttons. The user is then asked to randomly click the four buttons. We assigned 1 to two of the buttons, and 0 to the other two. After each click, the bits assigned to each button are changed. We have 6 permutations of bits. Also, to avoid having the user just click the same button all the time, we ignore input if the button just clicked is also the last button clicked.

We quickly learned that we had too few buttons, and too few permutations. Often, the user would just alternate between two buttons, even if he was made aware of the problem. This obviously leads to short cycles, which isn't very healthy. We wrote another program, with 9 buttons. Now we don't assign a bit to the last-clicked button. We have over 50 permutations of bits now. Although the user still is far more likely to click a button close to the last one he clicked, it doesn't matter as much now, as the cycles are much longer, and the user usually breaks the cycle before completing it.

We started with inbuilt de-skewing, but as it is really annoying (maybe even unhealthy) to use the program, we commented it out to minimise the time used.

The third program, "time," asks for user input in the command-window, and measures the time of the users reaction in milliseconds. The result is then stored modulo 2. At first, one was required to write a number before hitting enter, because we found out that some users tend to get into a rhythm, where he uses the same time for each keystroke, within about 3 milliseconds. That made the program a lot slower though, and we decided to just inform the user of the problem and ask him to vary his response time – it seemed to work.

The fifth program, "mousetime," is a mixture of "mouse" and "time." Basically, we reused the "mouse" program, and added a time factor, which we added to each bit. At first, we took the time between two clicks

(modulo 2,) but for some unknown reason, this was really biased towards 0 – about 80% of the time bits were 0. We changed it to just measure the system time modulo 2, and the problem was solved.

The last program, “all,” does the same thing as “mousetime,” but adds a third bit from files created with “typing.” We know get input from 3 different sources, and should end up with a pretty good result. We could further try to improve the program by not using XOR on the three input bits, but instead append the three sequences and use some hash-function (like SHA-1,) but again we were stopped by time constraints.

The tests of our outputs thought us one thing: Generating truly random isn't very easy. Our early programs, mouse, time and typing all failed on 2 or 3 of 10 inputs, the improved mouse failed only on 1. However, the “better” mousetime failed on 4 of 10 inputs – we're still shaking our heads about that one, as we really didn't expect it to perform worse. The last program worked quite well, also failing on only one input.

So some of the programs, improved “mouse” and “all,” produced decent results. Given that one would expect a random sequence to fail each of the five tests 5% of the time, 1 out of 10 is quite reasonable, and may even lead you to conclude that the programs provide random output. However, we also did tests on the combined results on each program (the about 1600 bits produced with each program.) Each of these tests failed miserably on the pokertest, and most also failed the runstest.

Further tweaking of the programs may have yielded better results. However, it would have made the programs more difficult to use. De-skewing and applying hash-functions both would have required more input bits.

Further reading

Most of the theory part was taken from chapter 5 of Handbook of Applied Cryptography by Menezes, van Oorshot and Vanstone, and can be found in at www.cacr.math.uwaterloo.ca/hac

More about pseudo-random number generation can be found in Knuth's The Art of Computer Programming vol2: Seminumerical Algorithms

Our test programs are located at ~haj/cryptoreport in the daimi file system.