Spring 2017

Making Random - Cryptography and the Generation of Random Sequences

Samuel Barton
University of Southern Maine

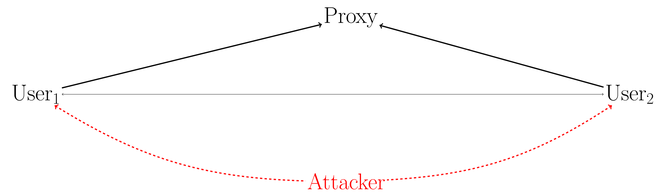
Making Random — Cryptography and the Generation of Random Sequences

Samuel Barton, Dr. David Briggs
University of Southern Maine

Background

It is surprisingly difficult to efficiently create sequences of numbers mathematicians and scientists regard as random, either by processes in the world, or algorithmically in software, yet many statistical experiments, cryptographic procedures, and computer games require randomly chosen values. The branch of cryptography dedicated to the generation of pseudorandom, or if possible genuinely random, sequences arose to address the need for vast quantities of values that are as random as possible.

One example of where random numbers are needed is the creation of keys for secured, confidential communication. The diagram to the right demonstrates this nicely. The protocol being shown there is RSA, where two end users generate private keys using (hopefully) random numbers, and then generate a public key using the key of a third party which both users have access to. The public key, when combined with either users private key, can decrypt the message. If an attacker were to determine the sequence of "random" numbers, then he or she might recreate the key and intercept the messages, modify them, or impersonate either user.



How the Algorithm Works

The algorithm utilizes the behavior of modern operating systems to generate random bits by putting the process to sleep for 10 microseconds, and then comparing the actual number of microseconds slept to the expected amount. The reason why this works for generating "random" bits is that modern operating systems do not wake processes exactly when they request. This is due to process scheduling. The algorithm uses the randomness in actual sleep time to generate each bit.

The bits are not truly random, as if a person were able to force the algorithm to always execute, then we would not have random bits. The other way to break the algorithm involves knowing absolutely everything about the system state and determining when a process would be woken up by the CPU scheduler. The bits generated by the algorithm will not cycle like the mathematical methods, and so form a software-based source of highly pseudorandom values. If we could generate genuine random numbers using mathematical methods, then we would never consider using a function like this which can at best generate $10^5 \frac{\text{bits}}{\text{second}}$, which is 10,000 times slower than the mathematical methods. Unfortunately, we cannot create truly random sequences using deterministic functions, i.e. functions that will generate the same results given the same inputs.

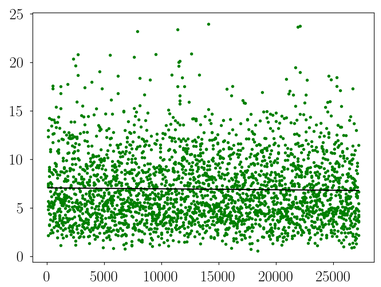
Testing

In order to validate the randomness the bits generated by the algorithm I ran three statistical tests against data produced by the algorithm. These tests come from Donald E. Knuth's book *The Art of Computer Programming — Volume 2: Seminumerical Algorithms.*

χ^2 Test

The first test I ran the algorithm against is the χ^2 test, where I generated 20 random bits with my algorithm and then summed them. This is analogous to flipping a coin 20 times. The probability distribution to compare to is the binomial distribution. With 20 degrees of freedom, we expect the χ^2 value to be between 15.45 and 23.83, this value comes from Donald E. Knuth's book. The mean of the χ^2 values my algorithm found over the span of tests done was 15.859.

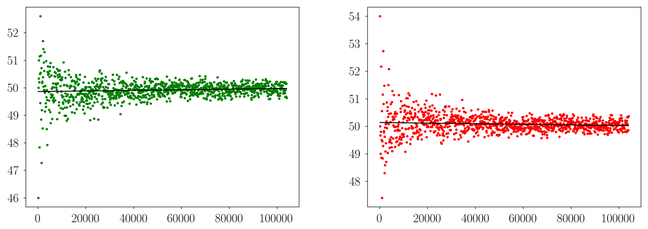
The graph's x-axis shows the number of times the algorithm was used to generate 20 bits and sum them, we'll call this N. The y-axis shows ten different χ^2 values calculated by taking ten different runs at N to show the spread in the results.



Frequency Test

The frequency test is designed to ensure that the data generated by the pesudorandom number generator is uniformly distributed. This algorithm should is expected to generate a uniform distribution of 0's and 1's.

The below two graphs show the frequency of 1's and 0's with the red graph being the 1's and the green graph the 0's. Both of these graphs behave as expected, where with a lower number of bits generated we have a larger spread of values, but as the number of bits generated gets larger we see that the frequencies converge to 50% for both 0 and 1. The mean of the frequency over all the tests is 49.92% for 1 and 50.08% for 0.



Algorithm

```
static int cur_dif = 0;

int time_dif()
{
    struct timeval start, stop;

    gettimeofday(&start, NULL);
    // sleep for 10 microseconds
    usleep(10);
    gettimeofday(&stop, NULL);

    int tmp = stop.tv_usec - start.tv_usec;
    if (cur_dif == tmp)
        return 0;
    else
    {
        cur_dif = tmp;
        return 1;
    }
}

int gen_rand()
{
    return time_dif() ^ (random()&01);
}
```

Permutations Test

The permutations test checks for randomness by testing an algorithms ability to uniformly generate all the permutations of length t if we generate n random numbers and divide them into t partitions. There are 2^t possible t-permutations of 0 and 1, and so the probability that a given t-permutation will be one particular partition is $\frac{1}{2^t}$. We multiply this probability by the number of permutations generated $\frac{n}{t}$ to get the expected number of times each partition should appear for a given n. I chose to make my partitions of size 3 and I test the algorithm's ability to uniformly generate all 8 possible 3-permutations of 0 and 1.

000 001 010 011 100 101 110 111

The graph below shows the χ^2 values for the 3-permutation calculations for various values of n. The expected χ^2, again from *The Art of Computer Programming: Volume 2*, for 7 degrees of freedom is between 4.255 and 9.037. The mean of the χ^2 values accross the total n tests is 6.941 which is right in the 50th percentile. Thus the algorithm generates these permutations with excellent uniformity.