Pseudorandom Number Generation

CS290G - Cryptographic Engineering

Johannes Omberg Lier
johannes.o.lier@gmail.com

John-Olav Storvold
johnolav.storvold@gmail.com

***Abstract*—A good pseudorandom number generator (PRNG) is an essential ingredient in creating a secure cryptographic system. The output numbers of those number generators are used futher in encrypting and decrypting messages. Since the process of generating these numbers is deterministic a security system will have a major security breach if those numbers produced are predictable either by having a predictable output pattern or simply do not have a "random" enough seed input for it's number generation. The goal is to have an easy to produce pseudorandom number generator than generates numbers as close to random as possible. There is essentially a tradeoff between having a complicated good PRNG and having a PRNG that is fast, but also secure enough to use in production.**

For our project we aim to create our own pseudorandom number generator which is fast in computation and generates reasonable results to be used in production. Our PRNG will have seed input gathered from various sources. The input seed will not be random, but will have an unpredictable pattern. Our motive is not to come up with a groundbreaking algorithm that exceeds already implemented algorithms, but to see if we can come up with an algorithm that is good enough to obtain some level of security.

When we have a working PRNG we will test the produced numbers and see if we have managed to create an algorithm that produces close to statistically random numbers.

I. Introduction

A pseudorandom number generator is used in a wide range of applications and methods. It is found most commonly in video games where content is generated automatically by an algorithm rather than by a game designer or programmer, in various scientific simulations and last but not least in cryptographic applications that require the outputs to be unpredictable compared to earlier outputs.

Our goal with our pseudorandom number generator is to make is as close to statistically random as possible as well as keeping the algorithm fast. To test if our generator is statistically random we will implement some of the diehard tests developed by George Marsaglia[1][4].

A. The diehard tests

The Diehard tests are statistical tests designed to measure the quality of random number generators. The battery of tests consists of fifteen different tests, designed to test different aspects of random. The more tests we perform the more accurate the results will be, but we think it is sufficient to implement two of them to test our pseudorandom number generator.

II. Implementation

For our implementation of the pseudorandom number generator we have decided to base our algorithm of the xorshift* algorithm which is build of the xorshift algorithm[2]. The xorshift algorithm generates the pseudorandom numbers by taking the exclusive or of a number with a bit shifted version of itself several times. This algorithm is both very efficient (extremely fast, according to Marsaglia) and takes up less space than other methods. The downside is that the input number needs to be chosen such that the period of the series of output is as long as possible. The period is the number of output numbers generated in a sequence before the same sequence occurs again.

The xorshift* algorithm takes an xorshift generator and does additional operation on the output. It takes the output modulo the size of the word as a non-linear transformation.

A. Discussion

It has been shown that Marsaglia's xorshift RNGs is quite similar to the linear feedback shift register (LFSR) generator[3]. In fact, with the right inputs the xorshift creates an indentical sequence compared to the LFSR. Performance wise, the xorshift is considered better both faster and generates better result in general.

III. Tests

As mentioned earlier, we have chosen two of the fifteen diehard tests.

- **The Runs test:** Create a sequence of length n, with randomly generated numbers. Count the number of ascending subsequences, s^+, and the number of descending subsequences, s^-. The ratio between the two should be close to $\frac{s^+}{s^-} = 1$.
- **Minimal Distance:** Choose $n = 8000$ random points in a 10000×10000 square. Find the minimum distance, d, between the $\frac{n^2-n}{2}$ pairs of points. The test is passed if the square of the minimum distance, d^2, is suffieciently close to being exponentially distributed with mean 0.995.

A. Results

1) The Runs test: We ran the Runs test 10000 times with the length of the sequence, $n = 10000$. For each time we ran the test we calculated the ratio $\frac{s^+}{s^-}$ for a randomly generated sequence. After we have calculated the ratio between the number of ascending subsequences and the number of descending sequences 10000 times, we calculated the average ratio. The

result we got was an average ratio of 1.0000027. We compared our implementation with Java's built-in random generator by running the same test with the same sequence length 10000 times using Java's built-in Random. The result we got from this was an average ratio of 1.000001. Using JUnit for testing the execution time is displayed, so we include it as an addition to the test. Execution time for our implementation: 2.346s. And the execution time for Java: 2.384s.

2) Minimum Distance: Again, we ran the test 10000 times following the description of the test[4]. The mean minimum distance we got after the executions was $d^2 = 1.3997$. Exectution time for out implementation: 0.870s. And the execution time for Java: 0.904s.

B. Discussion

1) The Runs test: From this test the difference between the expected result and the actual result was quite small (< 1 ‰). Our implementation performs slighly worse than Java's built-in random generator, but again the difference is very small. We have not studied further the probabilty that the number of ascending subsequences is exactly the same as the number of descending subsequences, but by observing the ratio of individual tests (on only one sequence, not 10000) even with large $n \geq 1000000$ the ratio is surprisingly often exactly 1.0. The same goes for with Java's built-in random generator. We also found it curious that the ratio varied more with a low n. As for the execution times we assumes that no conclusion can be drawn because because of the small difference between them. The difference can have been caused background processes on our laptops or other variances in our implementation.

2) Minimum Distance: From the minimum distance test our result differs from the expected mean of 0.995. We express the difference like so:

$$1.3997 - 0.995 = 0.4047$$

$$\frac{0.4047}{0.995} = 0.4067 \approx 40.1\%$$

From this we can see that there is a difference of approximately 40.1 % from the expected mean. For comparison we ran the same test using Java's built-in pseudorandom generator. This gives us an idea of our difference is significant or not. Using the Java generator the mean of the square of the minimum distance were $d^2 = 1.4218$. We compare Java's build in pseudorandom generator with the expected mean:

$$1.4218 - 0.995 = 0.4268$$

$$\frac{0.4268}{0.995} = 0.4289 \approx 42.9\%$$

Compared to the mean we achieved, we get

$$|0.4047 - 0.4268| = 0.0221$$

$$\frac{0.0221}{0.4047} = 0.0546 \approx 5.5\%$$

In the minimum distance test our implementation has a smaller difference from the expected mean compared to Java's built in pseudorandom generator. The difference between the two implementations is approximately 5.5 %.

The same goes for this test when it comes to execution times; no significant difference.

IV. Conclusion

Two tests have been implemented and used on both our own implementation of a pseudorandom number generator and Java's built-in pseudorandom generator. In the first test, the Runs test, both algorithms did well and showed a result close to the expected result. In the original test described by Marsaglia, it states that the numbers generated should be floats in the range (0,1). Our version has random integers in no specific range. In the second test, the minimum distance test, our implementation did differ more significantly from the expected value of the mean of the square of the minimum distance. That being said, Java's implementation did slightly worse in our test. We measured the execution times of both tests for both implementations and there is no significant difference between them, so we will not discuss execution times further.

V. Summary

In this paper we have studied the implementation of a pseudorandom number generator, in particular the xorshift* algorithm. The algorithm is fast and use less memory space compared to other generators. We have described our implementation and we have tested and analyzed the results from the tests. We implemented two tests, the Runs test and the minimum distance test. Both described by George Marsaglia as a part of a battery of tests designed to test random generators. Our goal with this project was not to implement something better than what exists today, but rather to implement something "good enough" compared to other methods. That being said, based on the tests, we are happy with the result.

References

[1] George Marsaglia, *The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness*, 1995
[2] George Marsaglia, *Xorshift RNGs*, The Florida State University, 2003
[3] Richard P. Brent, *Note on Marsaglias Xorshift Random Number Generators*, Volume 11, Issue 5, Oxford University, 2004
[4] Balasubramanian Narasimhan, *JDiehard: An implementation og Diehard in Java*, Department of Statistics, Stanford University, 2001