Amin Nouiser (anouiser@kth.se)
CINTE

Background

Pseudo random number generators

A pseudo random number generator (PRNG), unlike a real random number generator does not generate real random numbers. Whereas the latter uses random sources as e.g atmospheric noise the first uses a predefined seed and algorithm to generate a predictable sequence of random numbers. Using the same seed and PRNG algorithm, the generator should produce the same sequence of numbers each time.

One algorithm used for PRNGs is the linear congruential generator algorithm (see *figure 1*). In this recursive algorithm, x_0 is the seed (key when en-/ decrypting), a the multiplier and b the offset and m the “threshold”.

$$x_{i+1} = (x_i a + b)\, mod(m);\ i \in [0, \infty],\ x_0 \in [-\infty, \infty]$$

figure 1

Implementation

As the implementation was to be used for byte stream encryption the PRNG implemented had to meet some requirements:

- The numbers generated should be in the range of [0x00, 0xFF]
- Long period: the same number should not be generated more than once during each period meaning that each number should be equally “possible”

In order to meet the above requirements, **m was set to 257, a to 3 and b to 0**. In this way, all numbers between 1 (including) and 256 (including) would be generated once each period. As 256 was not in the range of desired numbers it was simply filtered out using an if statement.

In order to test the class, the junit5 framework was used. The test used different seeds (keys) and invoked the next(bits int) method 255 times and checked that each number [1, 255] had been generated exactly once. It also checked that the seed value was set to its original value after each period. The seeds used were 200, 27 and 30.

The implementation was tested with 32 bit number generation. However, as too many zeros were generated after masking out the upper 24 bits the number of bits were set to at most 8.

Moreover was the java.util.Random version, nextInt(int bound) tested with the same seeds and the bits argument set to 8. The test showed that java.util.Random did not generate evenly distributed numbers. For example 190 was produced twice in a row when using 200 as seed and the bound argument set to 256.

It was found that some of the inherited methods from java.util.Random did not work, for example nextInt(int bits). Some backwards engineering was done, looking at the Random source code in order to fix this however in the end the issue still remained.

Amin Nouiser (anouiser@kth.se)
CINTE

Discussion

It is clear that the test conducted is not sufficient to mathematically prove correct functionality of the implementation. However it was sufficient to understand that the implementation functions as intended.

The method in the end still had some flaws, as for example being limited to 8 bit int generation and not working with java.util.Randoms nextInt(int bound) method. In order to solve these issues further backwards engineering has to be made.