How to crack a Linear Congruential Generator

Haldir[RET]

22nd December 2004

1 Introduction

The Linear Congruential Generator (LCG) is a common, but not secure way to generate random numbers for a given range.

$$Definition\ 1:\ x_n = ax_{n-1} + k_1\ modulo\ m\ for\ all\ n \geq 1\ and\ x_0 = k_0$$

Most common Pseudo Number Generators (PRNG) implemented in standard libraries use the LCG, so I presented in [1] a method to solve the linear Congruential Generator using recursive equations. The downside of that method was that you couldn't easily find the modulus used, either you knew it in advance or you had to try a few values, but you just needed three consecutive values to solve it. In this paper I will present a method which will solve all values of the LCG including the modulus with six or more consecutive numbers of PRNG output.

2 Mathematical Background

In [2] George Marsaglia analyzed Pseudo Random Number Generators. He found the flaw, that LCGs are mainly "falling into planes", this means that any three consecutive integers (x, y, z) with multiplier *a* fall on the lattice of points generated by all linear combinations of the three points $(1, a, a^2), (0, m, 0), (0, 0, m)$. Also any point (x,y) falls on the lattice generated by (1,a),(0,m). See [2] for more properties of this flaw. We use this to solve all three parameters of the LCG We take a list of consecutive integers:

$$207560540,\ 956631177,\ 2037688522,\ 1509348670,\ 1546336451,\ 429714088,\ 217250280$$

Four integers from the list form a 3x3 matrix:

$$\begin{pmatrix} 207560540 & 956631177 & 1 \\ 956631177 & 2037688522 & 1 \\ 2037688522 & 1509348670 & 1 \end{pmatrix}$$

Thanks to the flaws listed above and in the references, the determinant of this matrix will be an integer multiple of the modulus used in the LCG. The GCD of several more of these matrices should provide us with the real modulus m.
Since we now have m, we can easily set up a linear system of equations which solves $207560540 * a\ +\ k\ =\ 956631177\ modulo\ m$ and $956631177 * a\ + k\ =\ 2037688522\ modulo\ m$. The results are: $a = 16807, k = 78125, m = 2^{31} - 1$

3 Source Code

I implemented a proof of concept version with Victor Shoup's NTL library. The source is designed for prime moduli. The source also calculates the seed used in the LCG. The determinant and linear equation calculation is optimized for the small dimensions used here. For non-prime moduli you will often get small multiples of the modulus, that's why we factor the modulus, also a non-prime modulus might break linsolve().

```
/*
Short way for calculating the determinant of a 3*3 matrix
*/
ZZ calcDet(ZZ val1, ZZ val2, ZZ val3, ZZ val4)
{
return abs(val1*val3 - val1*val4 + val2*val3 - val2*val2 + val2*val4 - val3*val3);
}
/*
Just to remove a few factors, the modulus might still
multiplied by a small factor, like 2 or 10
*/
ZZ factor(ZZ value)
{
  unsigned int sprimes[16] = {0x02,0x03,0x05,0x07,0x0B,0x0D,0x11,0x13,
                              0x17,0x1D,0x1F,0x25,0x29,0x2B,0x2F,0x35};
  int counter = 0;
  while(counter < 16)
  {
  if(value % sprimes[counter] == 0) value /= sprimes[counter];
  else counter++;
  }
  return value;
}
/*
Should do the trick to calculate the different GCDs of the determinants
*/
ZZ getModulus(ZZ * detlist, int length)
{
ZZ * tempGCD = new ZZ[length];
for(int i = 0;i<length;i++)
{
tempGCD[i] = detlist[i];
}

for(int k = 1;k<length;k++)
{
for(int i = 0;i<length-k;i++)
```

```
{
tempGCD[i] = GCD(tempGCD[i],tempGCD[i+1]);
}
}
return factor(tempGCD[0]);
}
/*
Short way to solve the 2 equations
*/
void linsolve(ZZ val1, ZZ val2, ZZ val3, ZZ modulus, ZZ &retval1, ZZ &retval2)
{
ZZ_p::init(modulus);
ZZ_p temp1 = to_ZZ_p(val1);
ZZ_p temp2 = to_ZZ_p(val2);
ZZ_p temp3 = to_ZZ_p(val3);
retval1 = rep((temp2-temp3)/(temp1-temp2));
retval2 = rep(temp2-(to_ZZ_p(retval1)*temp1));
}
/*
This code calculates the Seed of the LCG if nth number is known
Just a cheap reversal of the LCG
*/
ZZ calcSeed(ZZ a, ZZ k, ZZ m, ZZ nr, int nth)
{
ZZ_p::init(m);
ZZ_p retval;
retval = to_ZZ_p(nr);
for(int i = nth-1;i>=0;i--)
{
retval -= to_ZZ_p(k);
retval /= to_ZZ_p(a);
}
return rep(retval);
}
int main()
{
int outputcount = 8;
int detlistlength = outputcount - 3;
ZZ * lcgoutput = new ZZ[outputcount]; // lcgoutput contains
// the collected consecutive lcg output values
ZZ * detlist = new ZZ[detlistlength];
ZZ modulus;
ZZ a,k,seed;
cout << "Using the LCG values: " << endl;
/* Basic Linear Congruential Generator */
```

```
__int64 result = 12345;
for(int i = 0;i<8;i++)
{
result = 16807*result+78125;
result %= 2147483647;
lcgoutput[i] = result;
cout << result << " ";
}
/* -----------------------------------*/
cout << endl;

for(int i = 0;i<detlistlength;i++)
{
detlist[i] = calcDet(lcgoutput[i],lcgoutput[i+1],lcgoutput[i+2],lcgoutput[i+3]);
}
modulus = getModulus(detlist,detlistlength);
for(int i = 0;i<outputcount;i++)
{
if(modulus < lcgoutput[i])
{
cout << "Modulus is too small (" << modulus <<"), probably no LCG" << endl;
exit(-1);
}
}
linsolve(lcgoutput[0],lcgoutput[1],lcgoutput[2],modulus,a,k);
cout << "The LCG function is : x[n] = ( " << a;
cout << "*x[n-1] + " << k << " ) % " << modulus << endl;
seed = calcSeed(a,k,modulus,to_ZZ(lcgoutput[outputcount-1]),outputcount);
cout << "The Seed used was: " << seed << endl;
return 0;
}
```

4 Future Considerations

We can solve simple LCGs now, given only a handful of consecutive output values. Now it's time to think about a modification of the LCG. A very common one is to shift the output to obtain values of a certain size or to remove weak low order bits. For example Microsoft Visual C Runtime's rand() does a shift right of 16 bits together with an AND 0x7FFF. Since this just leaves us 15 bits of output to work with, it might not be possible to recover the seed, especially if the shift is even larger than 16 bits. Choosing the right parameters for the LCG is a bigger problem as it might sound, especially the multiplicand a is very important for the quality of the PRNG, the addend k is of minor importance. You should always run a selection of statistical tests before you choose a certain LCG.

References

[1] `http://www.reteam.org/`

[2] Marsaglia,G. (1968). Random Numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences,* **61**,25-28