Chosen sequences among linear congruential generators using SAT/SMT solvers: a Valentine's Day card to Mayflor Jamero Holden

William John Holden

February 11, 2020

What does the following program do?

```
import java.util.*;

public class RandomMessage {

    private static long[] l = new long[] {
        247057036367419L, 35543432715160L, 253980514236592L,
        89613466789088L, 53058416132968L, 111106152524424L,
        231264589651259L, 218141079436269L, 203524665566275L,
        99273655041031L, 217249958230482L
    };

    public static void main(String args[]) {
        for (int i = 0 ; i < l.length ; i++) {
            Random random = new Random(l[i]);
            System.out.print((char)(random.nextInt() & 95));
            System.out.print((char)(random.nextInt() & 95));
            System.out.print((char)(random.nextInt() & 95));
        }
    }
}
```

You probably suspect that it does something unexpected, and of course you are correct. This program iterates through each long integer listed in the `l` array, instantiates a new pseudo-random number generator (PRNG) with that value as a *seed*, and prints the PRNG's first three outputs as uppercase letters.

Before we look into the particular output, we should quickly review the internals of the `java.lang.Random` class. The `Random` class implements a linear congruential formula defined by D. H. Lehmer and described by Donald Knuth in *The Art of Computer Programming*. The OpenJDK implementation for `Random` can be found at `http://hg.openjdk.java.net/jdk10/jdk10/jdk/file/777356696811/src/java.base/share/classes/java/util/Random`.

java. Inside this implementation we see three magical constants (multiplier, addend, and mask) and an instance variable seed of type AtomicLong.

When the constructor for Random is provided a seed value, this initial seed is first scrambled by the initialScramble method. initialScramble performs a bitwise exclusive or against the multiplier constant, followed by a bitwise and against the mask. Random provides no hints for the origin of the multiplier value, which is 0x5DEECE66DL (decimal 25214903917).

By contrast, the origin for mask is clear. Random is designed for a 48-bit seed. mask is initialized with the statement (1L << 48) - 1), which results in bit string of 16 zeros and 48 ones.

Random contains a protected next method that computes the next seed as nextseed = (oldseed * multiplier + addend) & mask and returns up to 48 bits of the new seed. The public nextInt method is implemented as the one-liner return next(32).

Seeing the beautiful simplicity of this program, can we construct a seed to produce chosen sequences? Yes. In Dennis Yurichev's *SAT/SMT by Example*, we see that it is possible for Microsoft Visual C++'s (MSVC) rand function to produce eight consecutive decimal numbers ending with zero. Moreover, Yurichev shows a method to discover the necessary seed value using Microsoft's Z3 SAT/SMT solver.

The following Python program builds upon Yurichev's previous work and discovers the seed values necessary to recreate a desired string.

```
from z3 import *

def find_message_seed(msg, start):
    # Instantiate a new Z3 solver.
    s = Solver()

    # Java's PRNG uses the following three constants.
    multiplier = BitVec('multiplier', 64)
    mask = BitVec('mask', 64)
    addend = BitVec('addend', 64)
    s.add(multiplier == 0x5DEECE66D, mask == (1 << 48) - 1, addend == 0xB)

    # The seed is the answer we are looking for.
    # To word backwards from our intended message to the seed, the PRNG
    # will have to cycle through one or more states.
    # I kind of doubt we will ever find a lucky enough streak of PRNG
    # results to get past eight states.
    seed = BitVec('seed', 64)
    states = [BitVec("state{n}".format(n = i), 64) for i in range(0,7)]

    # Again, we don't know the seed, but we do not know the seed will be
    # used to generate the initial state of the PRNG. Also, yes the
    # multiplier gets "anded" to the seed, not multiplied.
    # http://hg.openjdk.java.net/jdk10/jdk10/jdk/file/777356696811/src/java
```

```
    .base/share/classes/java/util/Random.java#l145
s.add(states[0] == (seed ^ multiplier) & mask)

# Specify state transitions. This reproduces the behavior of
# http://hg.openjdk.java.net/jdk10/jdk10/jdk/file/777356696811/src/java
    .base/share/classes/java/util/Random.java#l203
# Note that the magic value "addend" is added here but was
# not used to get to state[0] above.
for j in range(1,7):
    s.add(states[j] == (states[j - 1] * multiplier + addend) & mask)

# Z3 also needs to know that we are trying to relate the PRNG
# states to letters. For this, we will use up to seven characters.
# Note that state 0 does not correspond to a character. State 0 is
# the initalized state that can never be directly used to generate
# a returnable value from the PRNG.
characters = [BitVec("ch{n}".format(n = i), 64) for i in range(1,7)]

# Ok, we've got our PRNG states and characters. Now we need to marry
# them somehow. In this program, the relation between a PRNG state
# and a character is that we:
# 1) Shift the seed by 16 bits just like java.util.Random.next(32),
# 2) Extract only the last seven bits from this value using a mask,
# 3) Always set the fifth bit (0x20) to make the letter uppercase.
#
# This program can handle spaces.
# Leaving the fifth bit free will allow you to work with more
# characters, but you might not be able to find solutions for your
# input.
#
# I tried making this an "or" statement so that the solver could
# branch with the fifth bit set or unset. Bad idea.
for k in range(0,6):
    s.add(characters[k] == ((states[k + 1] >> 16) & 0x7f) | 0x20)

# Now we can constrain the value for these characters.
# We walk from left to right, adding constraints letters to our
# string until the solver determines no such sequence is possible.
longest_seed = -1
i = 0
# You don't have to constrain i to just 3. In my experience, though,
# you won't find many outputs that produce more than 3 characters,
# while you can fairly reliably find three-character sequences.
# The program completes much faster with this constraint.
while i < 3 and start + i < len(msg):
```

```python
        s.add(characters[i] == ord(msg[start + i]))
        if s.check() == sat:
            i = i + 1
            longest_seed = s.model()[seed].as_long()
        else:
            break

    # Return the value of the longest seed as a long with the
    # number of characters constructed.
    return (longest_seed, i)

def find_message_seeds(msg):
    position = 0
    result = []
    while position < len(msg):
        (seed, msg_length) = find_message_seed(msg, position)
        result.append((seed, msg_length))
        position += msg_length
    return result
```

So what does the Java program on the first output?

```
$ java RandomMessage
HAPPY VALENTINES DAY MAYFLOR LOVE
```

Happy Valentine's Day, Mayflor!