Wyh Ranmdnoses Mattres

Frans Lategan

About me

- Principal Security Consultant at Aura
- github.com/fransla
- @fransla

The bascis

- What is meant by random?
  - You cannot predict the next output
  - Flat distribution

What are random numbers used for?

- Crypto(graphy)
  - Key generation
  - Seeds and nonces
- The other crypto(currency)
  - Address generation (* key generation)
  - Nonces, seeds, …
- Games of chance
  - The “chance” bit
- Other random things - session ids, GUIDs, salts for passwords, concurrency delays,
- hopefully not OTPs :-)

Where do we get them from?

- Computers are deterministic
  - That is a fancy way of saying NOT random
- Pseudo random functions are used
  - Which is actually another fancy way of saying NOT random
- But at least it normally gives a flat distribution

```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

OK, more random than that please

- Linear Congruence Generator or LCG
  - Java's java.util.Random,
  - POSIX [ln]rand48,
  - glibc [ln]rand48[_r]
- More examples:
  - https://en.wikipedia.org/wiki/Linear_congruential_generator

next

`protected int next(int bits)`

Generates the next pseudorandom number. Subclasses should override this, as this is used by all other methods.

The general contract of `next` is that it returns an `int` value and if the argument `bits` is between `1` and `32` (inclusive), then that many low-order bits of the returned value will be (approximately) independently chosen bit values, each of which is (approximately) equally likely to be `0` or `1`. The method `next` is implemented by class `Random` by atomically updating the seed to

```
(seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1)
```

and returning

```
(int)(seed >>> (48 - bits)).
```

This is a linear congruential pseudorandom number generator, as defined by D. H. Lehmer and described by Donald E. Knuth in *The Art of Computer Programming*, Volume 2: *Seminumerical Algorithms*, section 3.2.1.

Parameters:

`bits` - random bits

Returns:

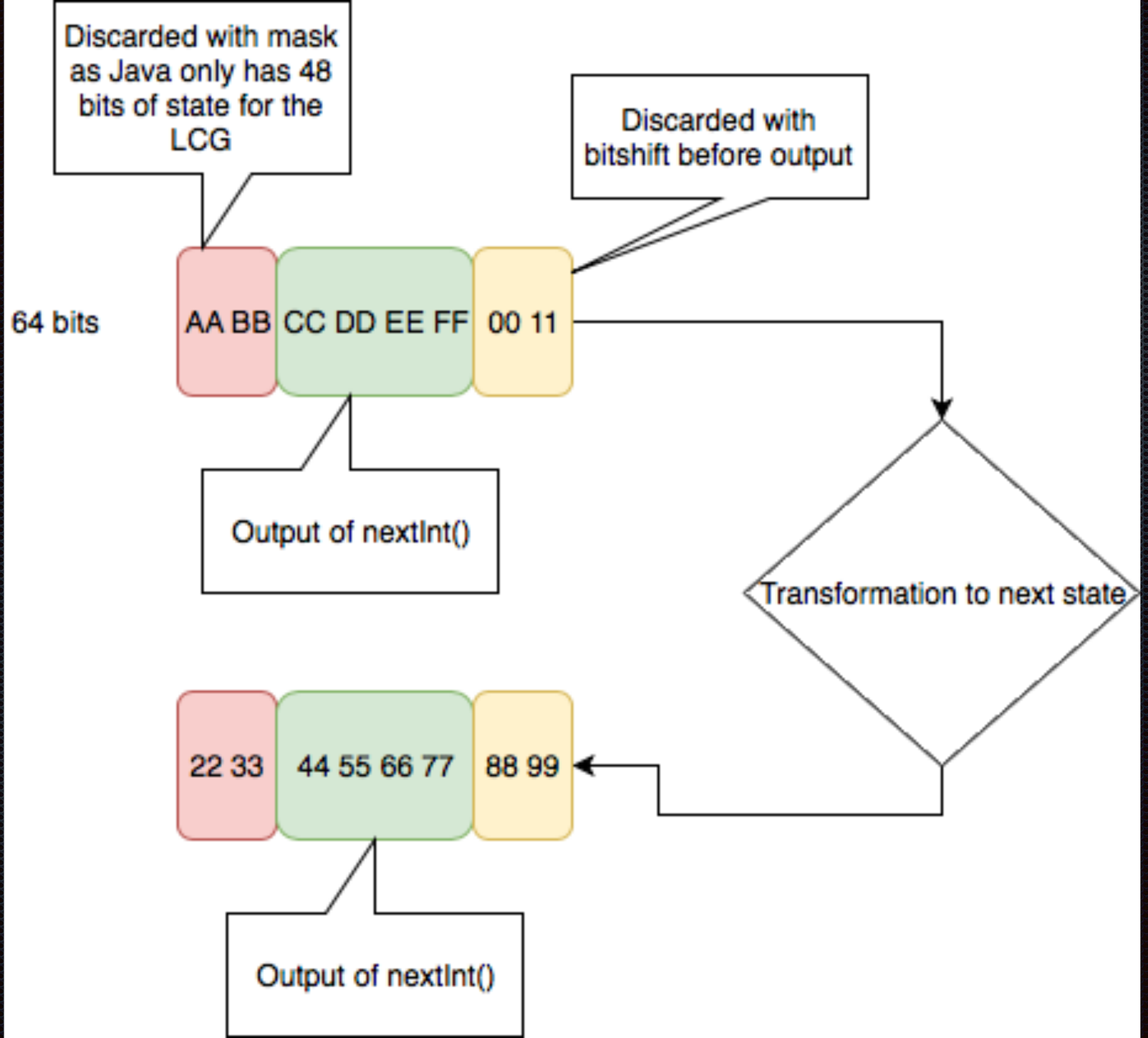
the next pseudorandom value from this random number generator's sequence

Since:

1.1

Teh Java LCG (1)

nextInt() calls next(32)


Discarded with mask as Java only has 48 bits of state for the LCG
Discarded with bitshift before output
64 bits
AA BB
CC DD EE FF
00 11
Output of nextInt()
Transformation to next state
22 33
44 55 66 77
88 99
Output of nextInt()

Levels. (Not random)

1. nextInt() (consecutive and missing values)
2. nextInt(n)
3. nextInt(n) but missing some values
4. nextInt(n) with n decreasing (Knuth or Fisher-Yates shuffle)

```
Random random = new Random();
long v1 = random.nextInt();
long v2 = random.nextInt();
for (int i = 0; i < 65536; i++) {
    long seed = v1 * 65536 + i;
    if (((seed * multiplier + addend) & mask) >>> 16) == v2) {
        System.out.println("Seed found: " + seed);
        break;
    }
}
```

Level 1: Hcaknig nextInt()

“Mode -> Dome -> Demo”

Failure is also an option

```
1   import java.util.*;
2
3   public class demo1 {
4     public static void main( String args[] ) {
5       Random seedgen = new Random();
6       int seed = seedgen.nextInt();
7       Random OTPGen = new Random(seed);
8
9       System.out.println("Nextint:");
10      for(int i=0; i<5; i++) {
11        System.out.println(OTPGen.nextInt());
12      }
13    }
14  }
```

Humans - not random

- While I set up the demo:
  - Think of a number between 1 and 10, inclusive.
  - Memorise it, there will be a test later

Demo 1 - nextInt

- ./randcrack_st -algorithm "LCG" -method “nextInt" -next 10 -missing 0 -values ""
- ./randcrack_st -algorithm "LCG" -method "nextInt" -next 10 -missing 3 -values ""

Humans - not random

7

Humans - not random

- 1 and 10 are out, first and last are not random
- Even numbers are not random
- 5 is in the middle and not random
- 3 and 7 are “prime” choices, but 3 is too close to the start

nextInt

`public int nextInt(int bound)`

Returns a pseudorandom, uniformly distributed `int` value between 0 (inclusive) a
sequence. The general contract of `nextInt` is that one `int` value in the specified
produced with (approximately) equal probability. The method `nextInt(int boun`

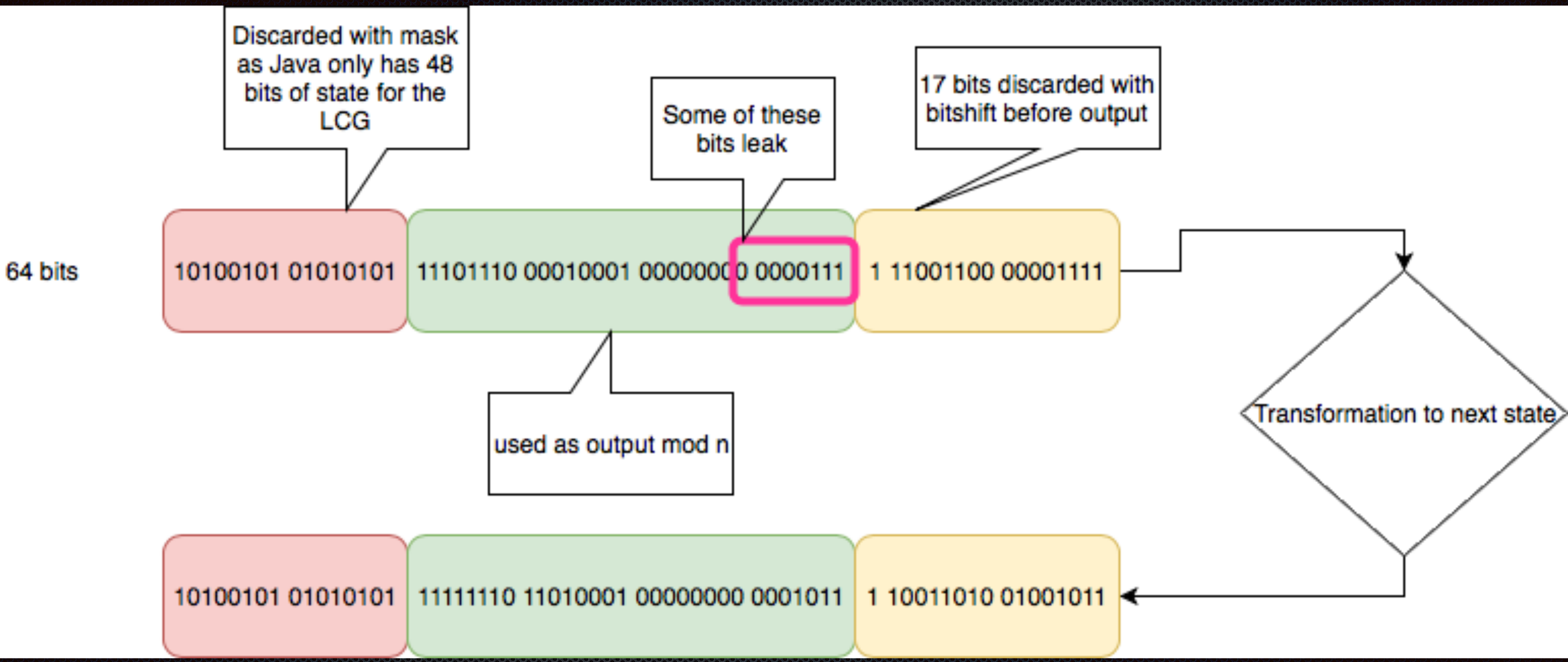
```
public int nextInt(int bound) {
  if (bound <= 0)
    throw new IllegalArgumentException("bound must be positive");

  if ((bound & -bound) == bound)  // i.e., bound is a power of 2
    return (int)((bound * (long)next(31)) >> 31);

  int bits, val;
  do {
      bits = next(31);
      val = bits % bound;
  } while (bits - val + (bound-1) < 0);
  return val;
}
```

Teh Java LCG (2)

nextInt(n) calls next(31) and mods it


Discarded with mask as Java only has 48 bits of state for the LCG
Some of these bits leak
17 bits discarded with bitshift before output
64 bits
10100101 01010101
11101110 00010001 00000000 0000111
1 11001100 00001111
Transformation to next state
used as output mod n
10100101 01010101
11111110 11010001 00000000 0001011
1 10011010 01001011

Bit leak for n values

- 36 - roulette - 2 bits
- 52 - cards - 3 bits
- 10000 - 4 digit OTP - 4 bits
- 48 - cards ignoring the first 4 - 4 bits

```
def crackNextIntn(values):
    """Tries to crack nextInt(n) given at least 2 values from the PRNG, with possibly missing values, n even but not a power of 2"""
    if len(values) < 2:
        return None
    toTest = values[1:]
    result = LCG()
    maxtest = (args.missing + 1) * len(toTest)
    seedlist = []

    # How many bits can we glean directly from the input i.e. if n is factored, how many times does 2 show up?
    n = args.probn
    directbits = prime_factors(n).count(2)
    paritybits = 2**(directbits) - 1 # so for 5 bits wil give 0b11111
    toTestbits = [x & paritybits for x in toTest]
    if args.verbose:
        print prime_factors(n)
        print directbits
        print paritybits

    #48 bits seed, low 16 (0-15) never shown, bit 16 also not output (see nextInt(n) - uses next(31))
    #so directbits are from bit 17 to bit 17+directbits.  Brute them, and check parity
    nomatch = False
    for lowseedbits in xrange(2**17):
        org = result.seed = (values[0] << 17) | lowseedbits
        nomatch = False
        index = lastFound = 0
        for j in xrange(maxtest):
            candidate = result.nextInt(n) & paritybits
            if candidate == toTestbits[index]:
                lastFound = j
                index = index + 1
            elif j > lastFound + args.missing:
                nomatch = True
                break
            if index == len(toTest):
                break

        if not nomatch:
            if args.verbose:
                print "found lower" , 17 + directbits, "bits %d" % org
            result.seed = org
            seedlist.append(org)
            if not args.full:
                break
    print seedlist
    for testseed in seedlist:
        if args.verbose:
            print "testing seed" , testseed
        result.seed = testseed
        # Now for the upper bits
        foundbits = 17 + directbits
        uppermask = 2**(foundbits) -1 # so for 22 bits wil give 0x7fffff
        tseed = result.seed & uppermask
        for uppercand in xrange(2**(48-foundbits)):
            result.seed = uppercand << foundbits | tseed;
            nomatch = False
            index = lastFound = 0
            for j in xrange(maxtest):
                candidate = result.nextInt(n)
                if candidate == toTest[index]:
                    lastFound = j
                    index = index + 1
                elif j > lastFound + args.missing:
                    nomatch = True
                    break
                if index == len(toTest):
                    break
            if not nomatch:
                result.seed = uppercand << foundbits | tseed;
                if args.verbose:
                    print "found! %d" % result.seed
                return result

    if args.verbose:
        print "Not found :-("
    return None
```

Level 2: Hcaknig nextInt(n)

Hcaknig nextInt(n)

- Determine the number of times 2 is a factor of n (i.e. only works if n is even. Also special case where n is a power of 2. Ignored for now)
- That is how many bits leak, starting with bit 17 (0 based)

Hcaknig nextInt(n)

- Brute the low 17 bits (0 - 16), apply the LCG transform, and check against the next value. Repeat the check for all the values we have. Keep valid candidates for the low 17 bits.
- Add the leaked bits (at least 1 :-)). For each candidate seed, brute the (at most) top 30 bits. Check the values and keep the valid ones.

“Mode -> Dome -> Demo”

Failure is also an option

```java
import java.util.*;

public class demo2 {
  public static void main( String args[] ) {
     Random seedgen = new Random();
     int seed = seedgen.nextInt();
     Random OTPGen = new Random(seed);

     System.out.println("Nextint(10000):");
     for(int i=1; i<20; i++) {
       System.out.print(OTPGen.nextInt(10000) + ", ");
     }
     System.out.println();

  }
}
```

Demo 2 - nextInt(n)

- ./randcrack_st -algorithm "LCG" -method "nextIntn" -next 10 -missing 0 -probn 10000 -verbose -values ""
- Takes about a minute... or 3

Real world hits again

- What if we cannot get hold of consecutive values?

```
def crackNextIntn(values):
    """Tries to crack nextInt(n) given at least 2 values from the PRNG, with possibly missing values, n even but not a power of 2"""
    if len(values) < 2:
        return None
    toTest = values[1:]
    result = LCG()
    maxtest = (args.missing + 1) * len(toTest)
    seedlist = []

    # How many bits can we glean directly from the input i.e. if n is factored, how many times does 2 show up?
    n = args.probn
    directbits = prime_factors(n).count(2)
    paritybits = 2**(directbits) - 1 # so for 5 bits wil give 0b11111
    toTestbits = [x & paritybits for x in toTest]
    if args.verbose:
        print prime_factors(n)
        print directbits
        print paritybits

    #48 bits seed, low 16 (0-15) never shown, bit 16 also not output (see nextInt(n) - uses next(31))
    #so directbits are from bit 17 to bit 17+directbits.  Brute them, and check parity
    nomatch = False
    for lowseedbits in xrange(2**17):
        org = result.seed = (values[0] << 17) | lowseedbits
        nomatch = False
        index = lastFound = 0
        for j in xrange(maxtest):
            candidate = result.nextInt(n) & paritybits
            if candidate == toTestbits[index]:
                lastFound = j
                index = index + 1
            elif j > lastFound + args.missing:
                nomatch = True
                break
            if index == len(toTest):
                break

        if not nomatch:
            if args.verbose:
                print "found lower" , 17 + directbits, "bits %d" % org
            result.seed = org
            seedlist.append(org)
            if not args.full:
                break
    print seedlist
    for testseed in seedlist:
        if args.verbose:
            print "testing seed" , testseed
        result.seed = testseed
        # Now for the upper bits
        foundbits = 17 + directbits
        uppermask = 2**(foundbits) -1 # so for 22 bits wil give 0x7fffff
        tseed = result.seed & uppermask
        for uppercand in xrange(2**(48-foundbits)):
            result.seed = uppercand << foundbits | tseed;
            nomatch = False
            index = lastFound = 0
            for j in xrange(maxtest):
                candidate = result.nextInt(n)
                if candidate == toTest[index]:
                    lastFound = j
                    index = index + 1
                elif j > lastFound + args.missing:
                    nomatch = True
                    break
                if index == len(toTest):
                    break
            if not nomatch:
                result.seed = uppercand << foundbits | tseed;
                if args.verbose:
                    print "found! %d" % result.seed
                return result

    if args.verbose:
        print "Not found :-("
    return None
```

Level 3: Hcaknig nextInt(n) with missing values

Demo 3 - nextInt(n) (missing values)

- ./randcrack_st -algorithm "LCG" -method “nextIntn" -next 10 verbose -missing 2 -values ""
- Takes about a minute... or 3

Level 4: Let's suhlffe smoe crads

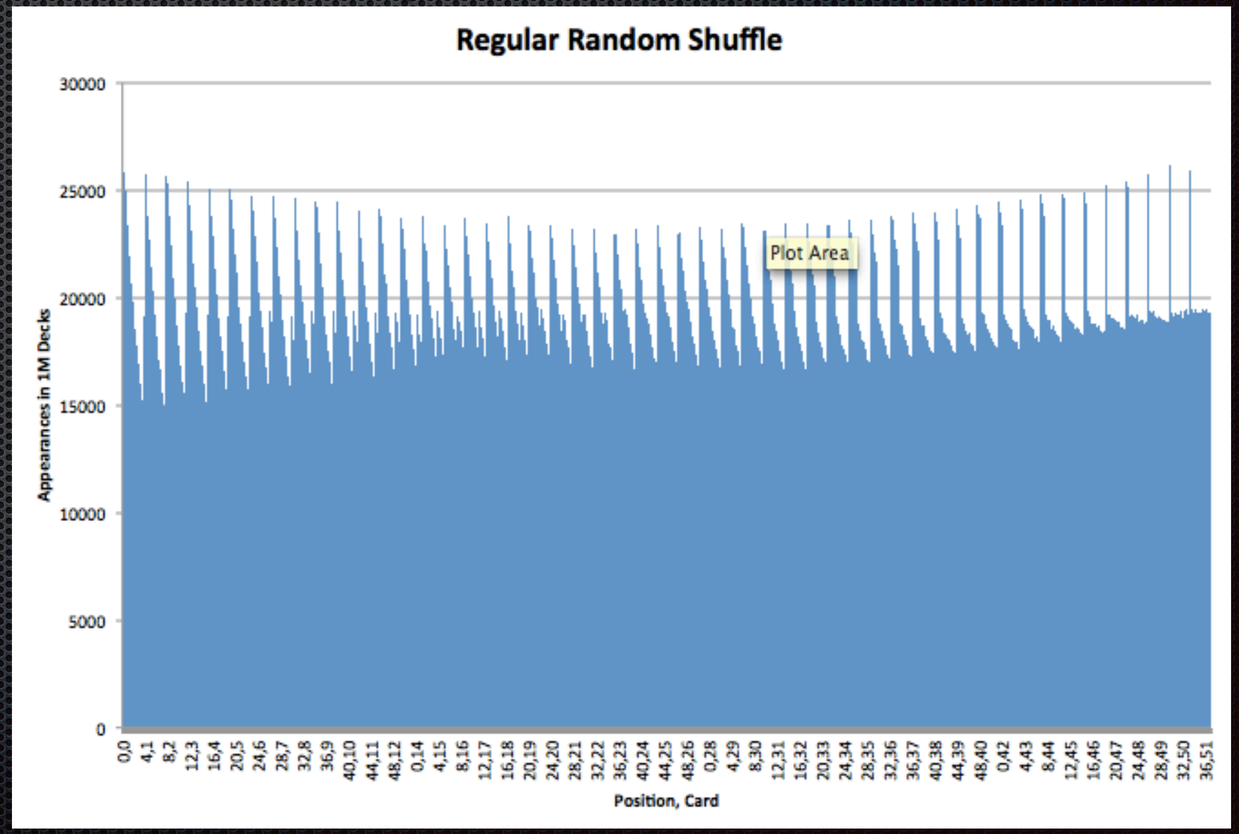
- Provably random, flat distribution.
- Knuth again (Fisher-Yates shuffle)
- But:
  - 52! ~ 2^226
  - Also, which random?

How not do it

```
1   Random rgen = new Random();  // Random number generator
2   int[] cards = new int[52];
3
4   //--- Initialize the array to the ints 0-51
5   for (int i=0; i < cards.length; i++) {
6       cards[i] = i;
7   }
8
9   //--- Shuffle by exchanging each element randomly
10  for (int i=0; i < cards.length; i++) {
11      int randomPosition = rgen.nextInt(cards.length);
12      int temp = cards[i];
13      cards[i] = cards[randomPosition];
14      cards[randomPosition] = temp;
15  }
```

Why not?

- Because the level 3 challenge would crack it :-)
- Also not a flat distribution



```
public static void shuffle(int[] arr) {
    Random r = ThreadLocalRandom.current();

    for (int i = arr.length - 1; i > 0; i--) {
        int index = r.nextInt(i + 1);

        int tmp = arr[index];
        arr[index] = arr[i];
        arr[i] = tmp;
    }
}
```

Semux cryptocurrency example

It is hraedr

- Leaked bits not constant, often 0
  - Means we need more values to crack
- Observing the effect (shuffled deck) a card moved twice will break the crack*
  - Means it will not always crack

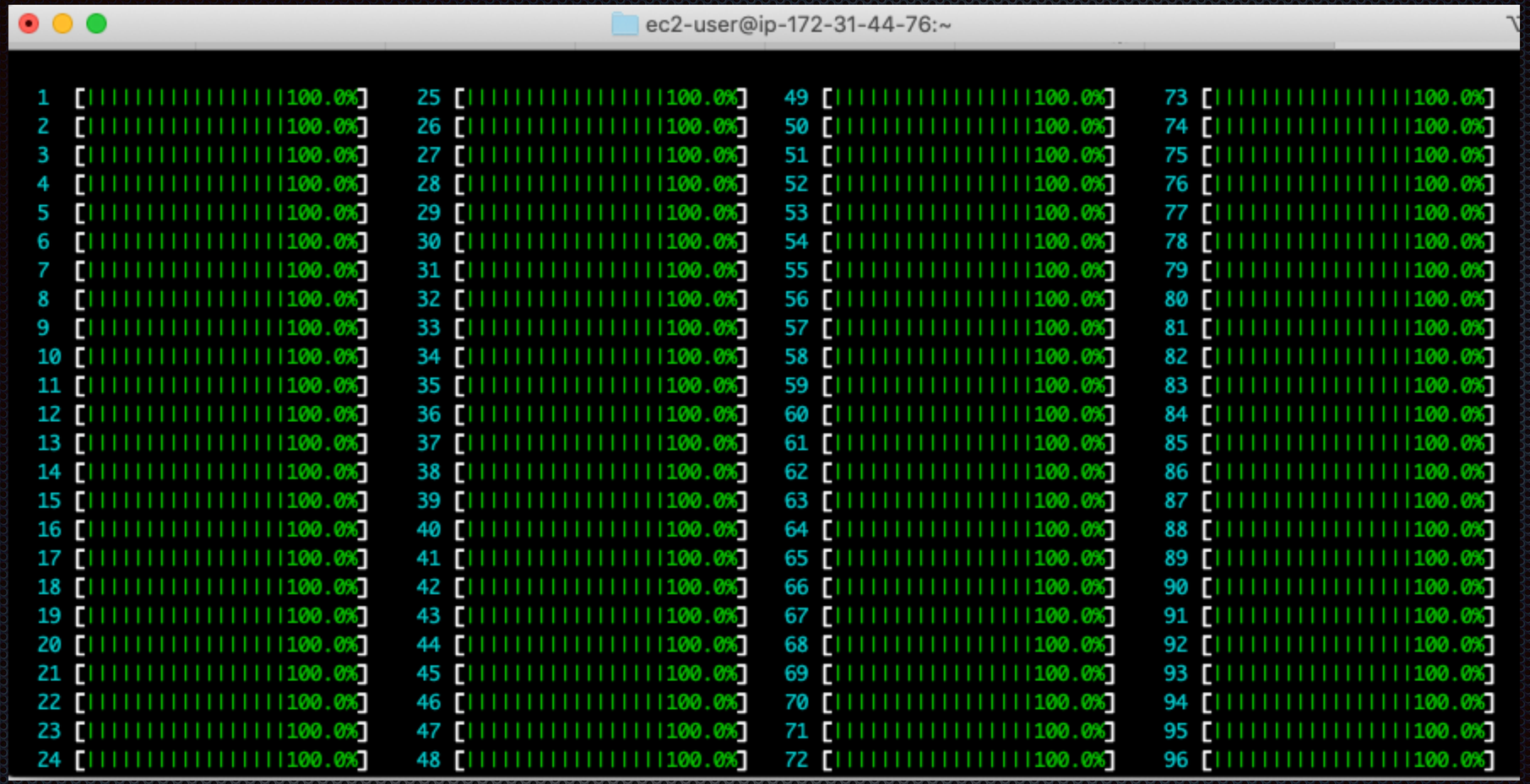
Demo 4 - nextInt(n), decrementing n

- ./randcrack_st -algorithm "LCG" -method "nextIntnDecr" -next 30 -missing 0 -probn 100 -values ""
  - 20 values, 2 min
- ./randcrack_st -algorithm "LCG" -method "nextIntnDecr" -next 30 -missing 1 -probn 52 -values ""
  - 18 values with a repeat/error, 5 mins*
- Takes about a 30s per seed

```
[ec2-user@ip-172-31-44-76 ~]$ time ./randcrack_mt -algorithm "LCG" -method "nextIntnDecr" -verbose -r
values "15, 42, 33, 27, 6, 24, 45, 28, 4, 19, 40, 25, 38, 11, 36, 14, 35, 9"
logger: randcrack_mt.go:401: LCG
logger: randcrack_mt.go:402: 15, 42, 33, 27, 6, 24, 45, 28, 4, 19, 40, 25, 38, 11, 36, 14, 35, 9
logger: randcrack_mt.go:280: Direct bits: 2
logger: randcrack_mt.go:281: Parity bit mask: [3 0 1 0 3 0 1 0 3 0 1 0 3 0 1 0 3 0]
logger: randcrack_mt.go:287: toTest [15 42 33 27 6 24 45 28 4 19 40 25 38 11 36 14 35 9]
logger: randcrack_mt.go:288: toTestbits converted to ints [3 0 1 0 2 0 1 0 0 0 0 0 2 0 0 0 3 0]
logger: randcrack_mt.go:294: maxtest 17
logger: randcrack_mt.go:321: 545 seeds to test
logger: randcrack_mt.go:333: Waiting To Finish
```

An example I prepared earlier...

time ./randcrack_mt -algorithm "LCG" -method "nextIntnDecr" -verbose -next 51 -missing 1 -probn 52 -values "15, 42, 33, 27, 6, 24, 45, 28, 4, 19, 40, 25, 38, 11, 36, 14, 35, 9"



An example I prepared earlier...

time ./randcrack_mt -algorithm "LCG" -method "nextIntnDecr" -verbose -next 51 -missing 1 -probn 52 -values "15, 42, 33, 27, 6, 24, 45, 28, 4, 19, 40, 25, 38, 11, 36, 14, 35, 9"

```
Success!
seed: 257537421547518. The next 51 values after 15 are:
nextint(51): 42, seed after: 128744014179121
nextint(50): 33, seed after: 234860954692072
nextint(49): 27, seed after: 247615503255507
nextint(48): 6, seed after: 131932978194146
nextint(47): 24, seed after: 156367225231941
nextint(46): 25, seed after: 237697597910380
nextint(45): 28, seed after: 173223196581639
nextint(44): 4, seed after: 43015467024390
nextint(43): 19, seed after: 19161156429465
nextint(42): 40, seed after: 281155459212592
nextint(41): 25, seed after: 75809677882747
nextint(40): 38, seed after: 16671734286186
nextint(39): 11, seed after: 105965501424685
```

An example I prepared earlier...

time ./randcrack_mt -algorithm "LCG" -method "nextIntnDecr" -verbose -next 51 -missing 1 -probn 52 -values "15, 42, 33, 27, 6, 24, 45, 28, 4, 19, 40, 25, 38, 11, 36, 14, 35, 9"

```
nextint(2): 1, seed after: 221470700436632
nextint(1): 0, seed after: 940602314333411

real    5m35.001s
user    531m53.839s
sys     3m24.938s
[ec2-user@ip-172-31-44-76 ~]$
```

An example I prepared earlier...

time ./randcrack_mt -algorithm "LCG" -method "nextIntnDecr" -verbose -next 51 -missing 1 -probn 52 -values "15, 42, 33, 27, 6, 24, 45, 28, 4, 19, 40, 25, 38, 11, 36, 14, 35, 9"

What does this mean?

- Practical and usable tool to crack Java's LCG
- If you use a weak PRNG, you will lose (money|fame|sleep)
- Important - no sign of hack in logs, can be 100% passive
- No direct success

Final tips

- Choose cryptographically strong random number generators for unpredictability
  - java.security.SecureRandom
- Check for the chosen distribution
  - Settlers of Catan - 2 D6 != 1 D12
- Code:
  - https://github.com/fransla/randcrack

End of code and demos

- Questions?
  - The answer is 42. 42. 42. 42.

Bbiloirgayph

- https://jazzy.id.au/2010/09/20/cracking_random_number_generators_part_1.html
- https://docs.oracle.com/javase/9/docs/api/java/util/Random.html#nextInt(int)
- http://crypto.stackexchange.com/questions/2086/predicting-values-from-a-linear-congruential-generator
- https://blog.cryptographyengineering.com/2015/01/14/hopefully-last-post-ill-ever-write-on/
- https://blog.cryptographyengineering.com/2017/12/19/the-strange-story-of-extended-random/
- Font: https://www.k-type.com/fonts/lexie-readable/
- http://chronicles.blog.ryanrampersad.com/2008/10/shuffle-an-array-in-java/
- https://random.org
- https://factorable.net/weakkeys12.extended.pdf