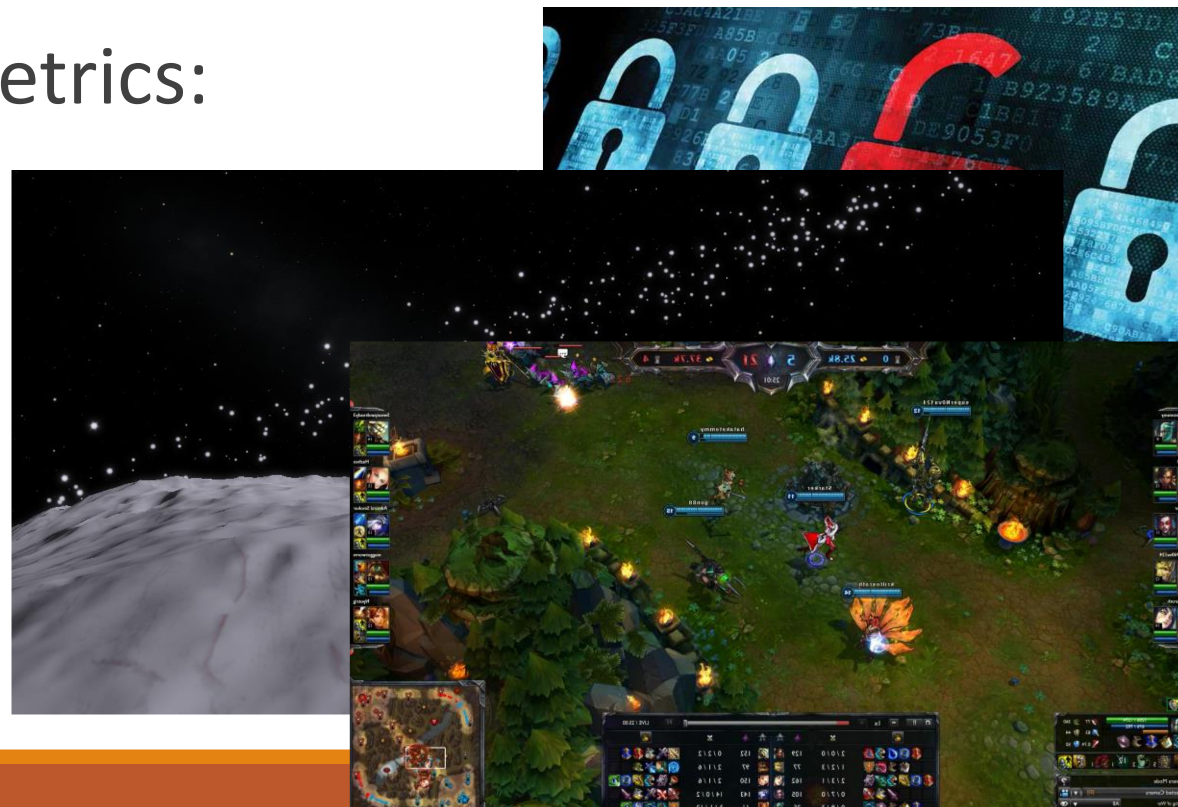
Dieharder Again:

Testing Modern Random Number Generators

KASHEV DALMIA & DAVID HUANG

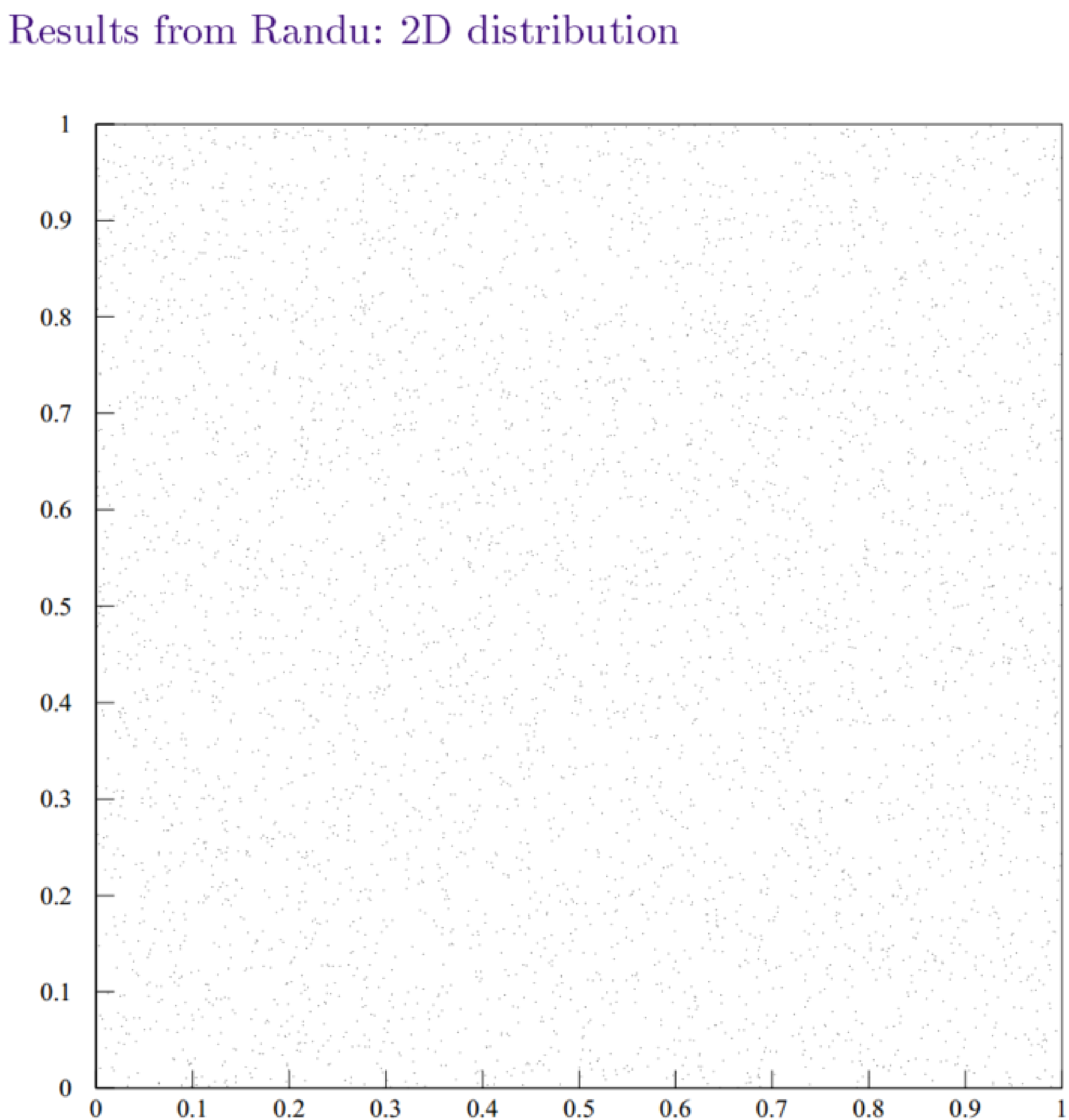
Why Study Random Numbers?

- Simulation, Cryptography, Gaming & More
- Quality Is Hard to Determine
- Important Metrics:
  - Randomness
  - Predictability
  - Memory Usage
  - Speed



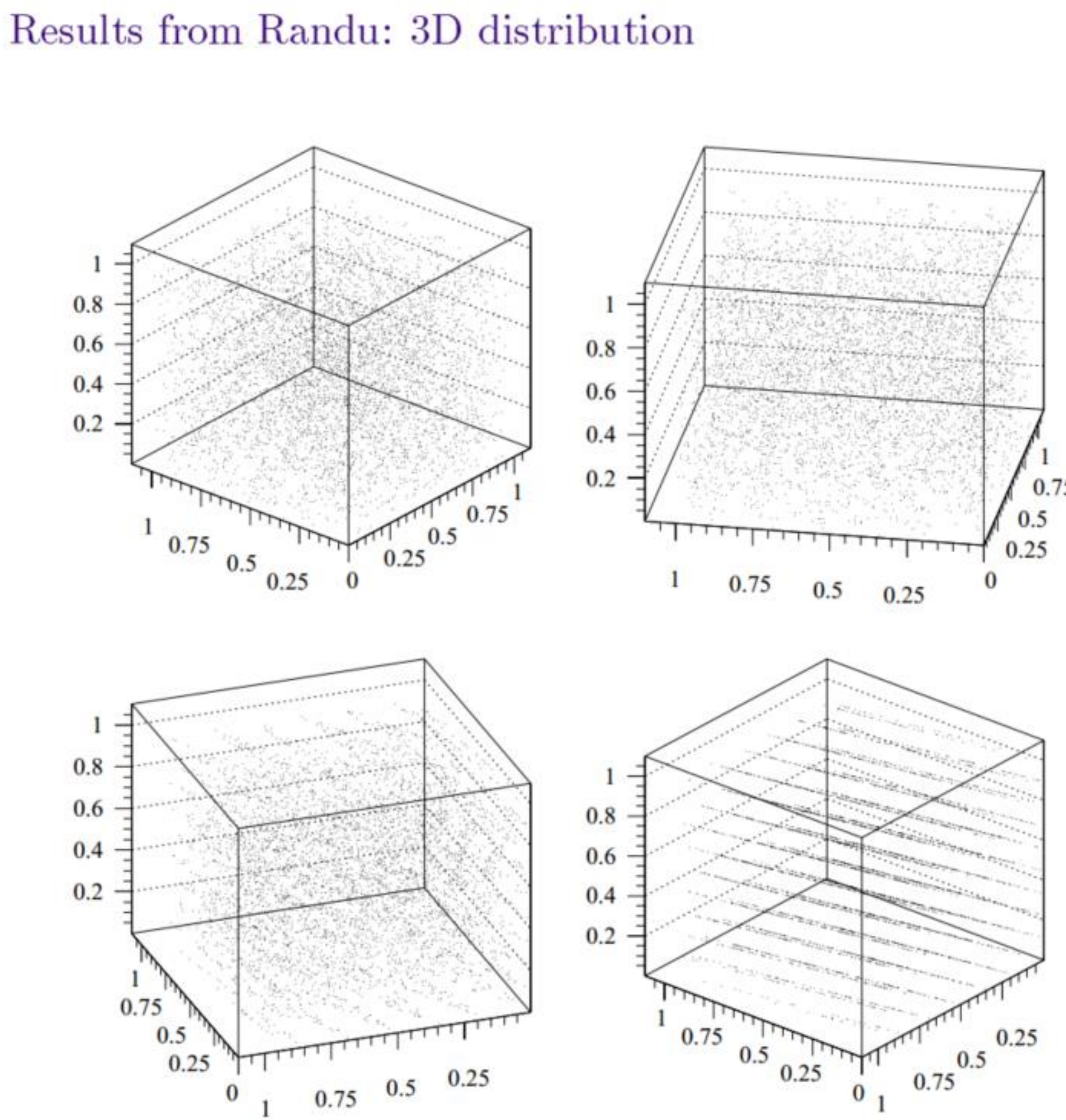
Why Test RNGs?

Example: RANDU



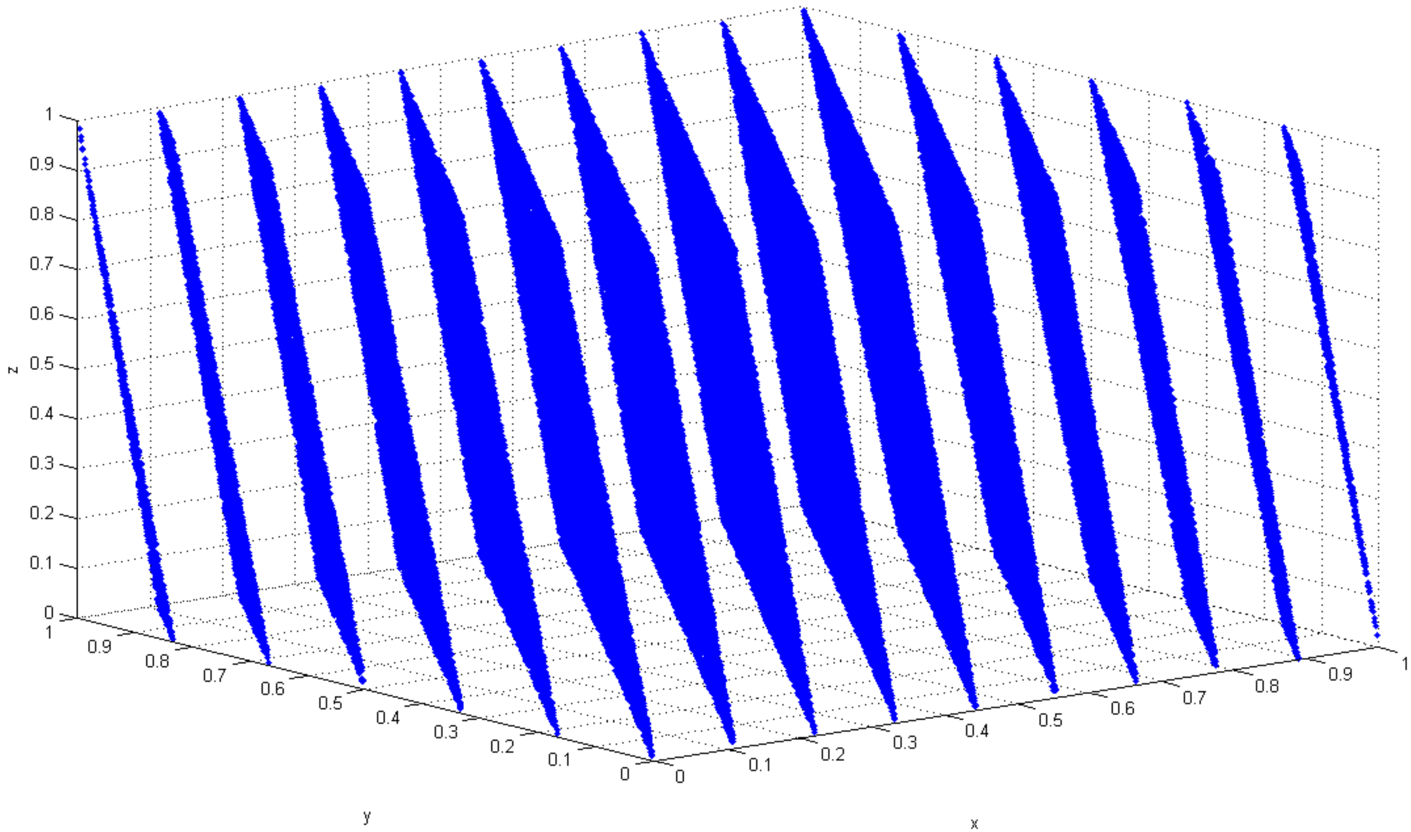
source: http://perso.ens-lyon.fr/eric.thierry/Perf2014/randu-issue.pdf

Example: RANDU



source: http://perso.ens-lyon.fr/eric.thierry/Perf2014/randu-issue.pdf

RANDU is Bad.



source: http://commons.wikimedia.org/wiki/File:Randu.png

Testing RNGs

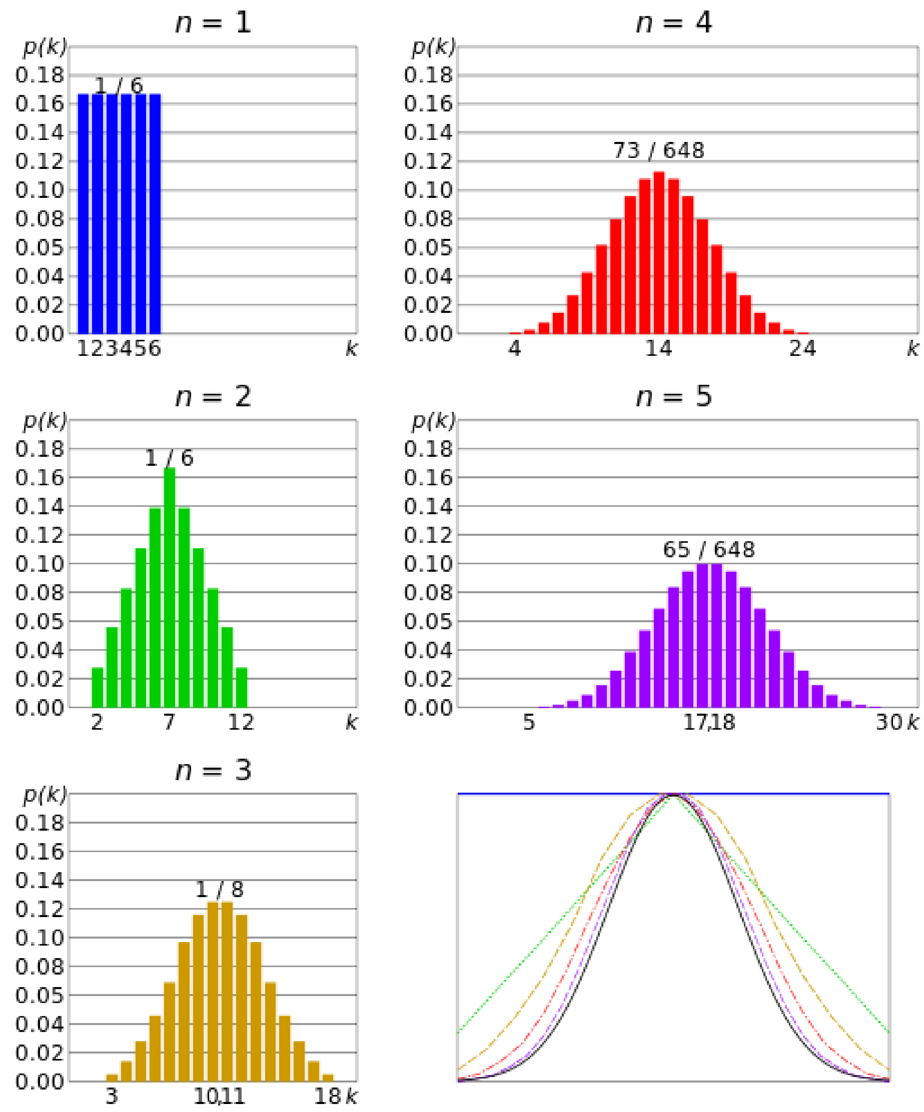
Dieharder: A RNG Testing Framework

- Suite of Randomness Tests
- Incorporates tests from Diehard, NIST, and more
- Low quality generators easily caught (RANDU)
- Available in Ubuntu Repositories
- Easy to interface with

How Dieharder Works

Example: Rolling a die and summing the result, repeatedly.

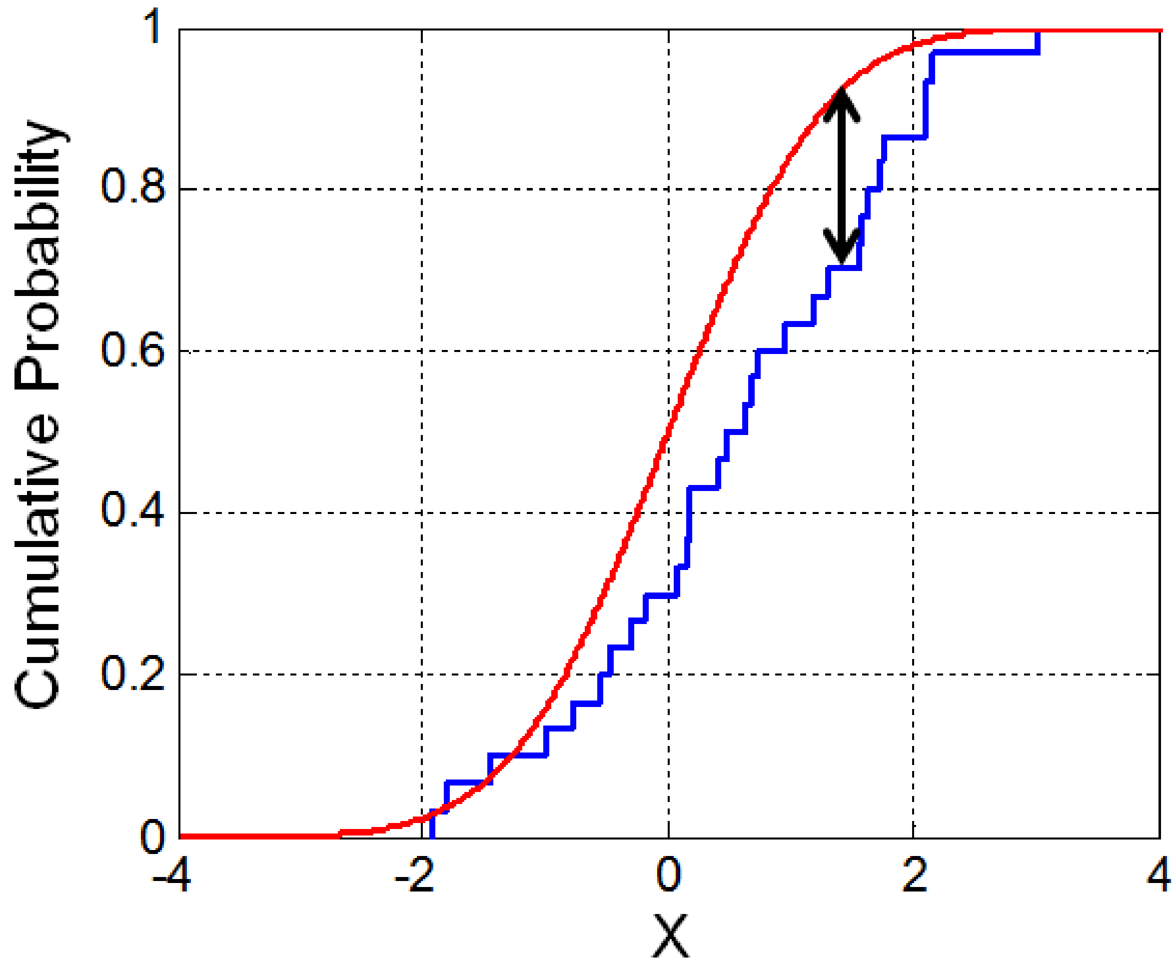
This creates a P-Value.



source: http://commons.wikimedia.org/wiki/File:Dice_sum_central_limit_theorem.svg

Kolmogorov -Smirnov Testing

How close are our empirical P-values to a perfect uniform distribution?



$$F_n(x) = \frac{1}{n}\sum_{i=1}^{n} I_{X_i \le x}$$

source: http://commons.wikimedia.org/wiki/File:KS_Example.png

Kinds of Tests in Dieharder

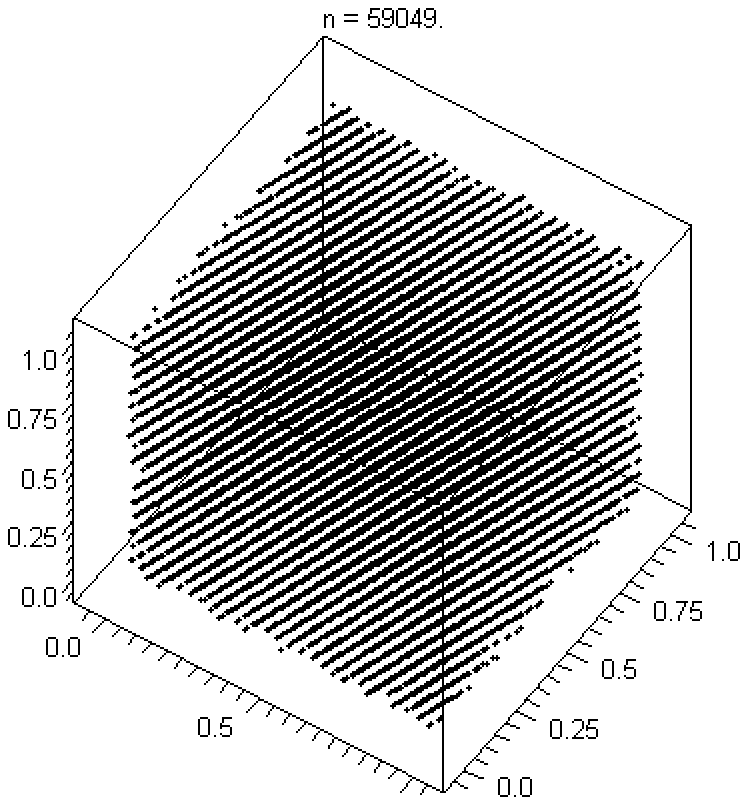
- Birthday Test – see if random Birthday distances in a year are exponentially distributed.
- Flip a Coin and ensure the coin is fair.
- Play games of Craps and ensure the average number of wins is distributed correctly.
- Throw away every *n* bits of output and ensure the remaining bits are uniformly distributed by summing them and ensuring a normal distribution.
- Count Runs of zeros and ones and ensure uniformity.

Kinds of RNGs

Linear Congruential

$$X_{n+1} = (aX_n + c) \bmod m$$

- MINSTD, RANDU
- **Strengths:** lightweight, small state, easy to understand
- **Weaknesses:** short period, poor quality planar output



source: http://commons.wikimedia.org/wiki/File:Lcg_3d.gif

Xorshift

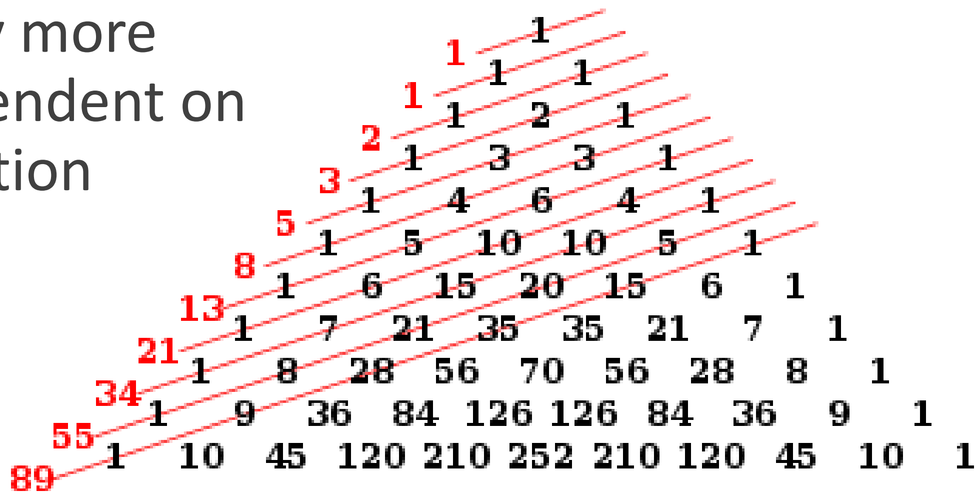
- Shift and XOR together past values
- **Strengths:** fast, small state, good period to performance ratio, somewhat good quality.
- **Weaknesses:** relatively short period, hard to chose parameters, some question of quality.

```c
// Random seed numbers
uint32_t t, x, y, w;
// Fixed constants for shifting
const uint32_t a, b, c;
uint32_t get_rand(void){
    t = x ^ (x << a);
    x = y;
    y = z;
    z = w;
    w = w ^ (x >> b) ^ t ^ (t >> c);
    return w;
}
```

Lagged Fibonacci

$$X_n = (X_{n-r} * X_{n-s}) \bmod m, \quad 0 < s < r$$

- **Strengths:** Flexible, can be fast
- **Weaknesses:** significantly more state, output quality dependent on seeding, parameter selection

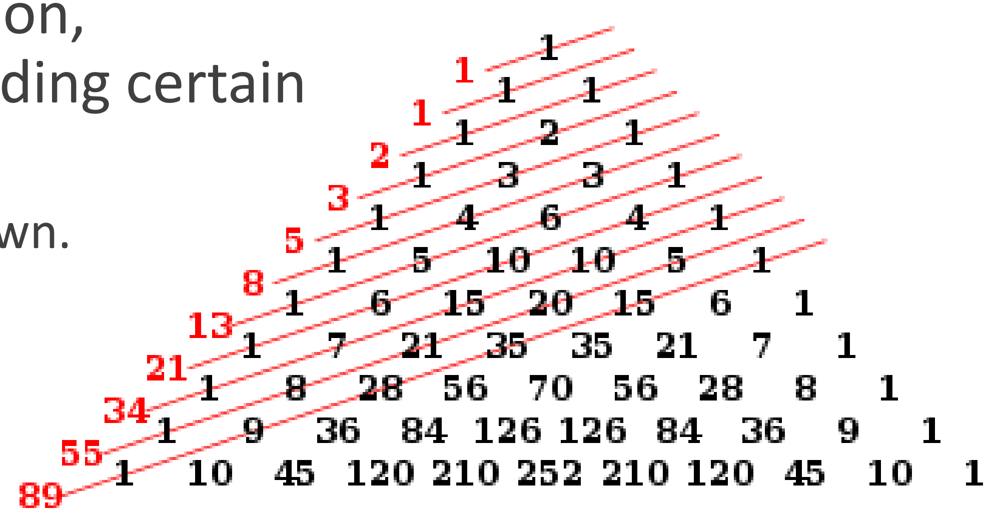


Lagged Fibonacci – Subtract With Carry & RANLUX

$$X_n = (X_{n-r} - X_{n-s} - c_{n-1}) \bmod m,\ 0 < s < r$$

$$c_n = 1 \text{ if } X_{n-r} - X_{n-s} - c_{n-1} \geq 0, \text{ or } 0 \text{ otherwise}$$

- RANLUX: reduce correlation, increase quality by discarding certain amount of output
  - Causes considerable slowdown.



[Complementary] Multiply With Carry

$$X_{n+1} = (aX_{n-r} + c_n) \bmod m$$

$$X_{n+1} = (b - 1) - (aX_{n-r} + c_n) \bmod m$$

- Period heavily dependent on parameter selection
- Proofs regarding period, seeding, parameter selection simpler in complementary version
- Huge periods possible with good speed.
- **Strengths:** long periods, fast, high quality output
- **Weaknesses:** Somewhat large state

Mersenne Twister

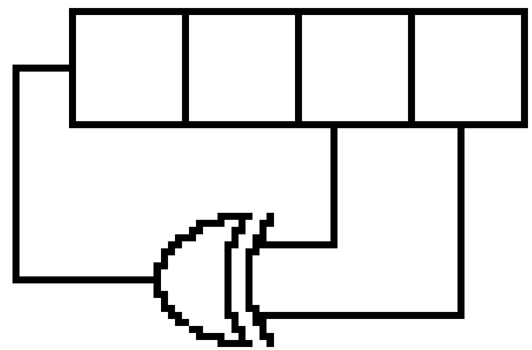
- Most common RNG in use
  - C++, Python, MATLAB, R, PHP, More
- **Strengths:** Very large period, extremely high quality, equidistributed output
- **Weaknesses:** complex operation and seeding, relatively slow, large state

Mersenne Twister Algorithm

$X_{k+n} = X_{k+m} \oplus (X^u_k \mid X^l_{k+1})A,$

$$A = \begin{bmatrix} \begin{matrix} 0 \\ \vdots \\ 0 \end{matrix} & I_{w-1} \\ a_{w-1} & a_{w-2} \cdots a_0 \end{bmatrix}$$

- Effectively a linear feedback shift register (LFSR) with transformed on each XOR operation
- $(X^u_k \mid X^l_{k+1})A$ concatenates two previous masked values, performs a right-shift, and applies an XOR mask on each bit



Mersenne Twister Algorithm, continued

- Tempering function on the state:

$$
\begin{aligned}
y &:= x \oplus (x \gg u) \\
y &:= y \oplus ((y \ll s) \cdot b) \\
y &:= y \oplus ((y \ll t) \cdot c) \\
y &:= y \oplus (y \gg l)
\end{aligned}
$$

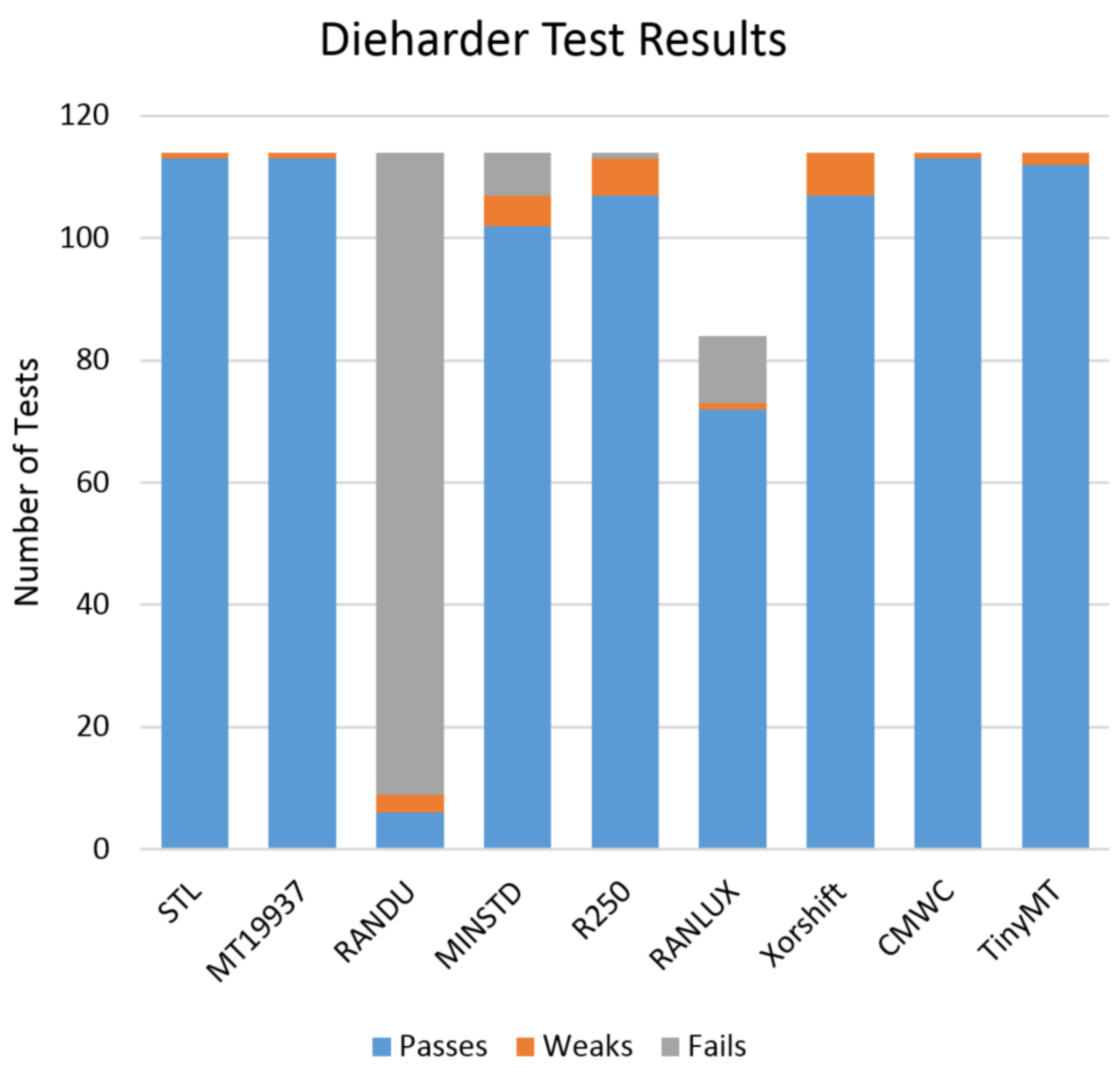
- Ensures equidistribution in output, to $d = 623$

- In practice, sequence is created in batch, then output is generated on request using the tempering function

Results

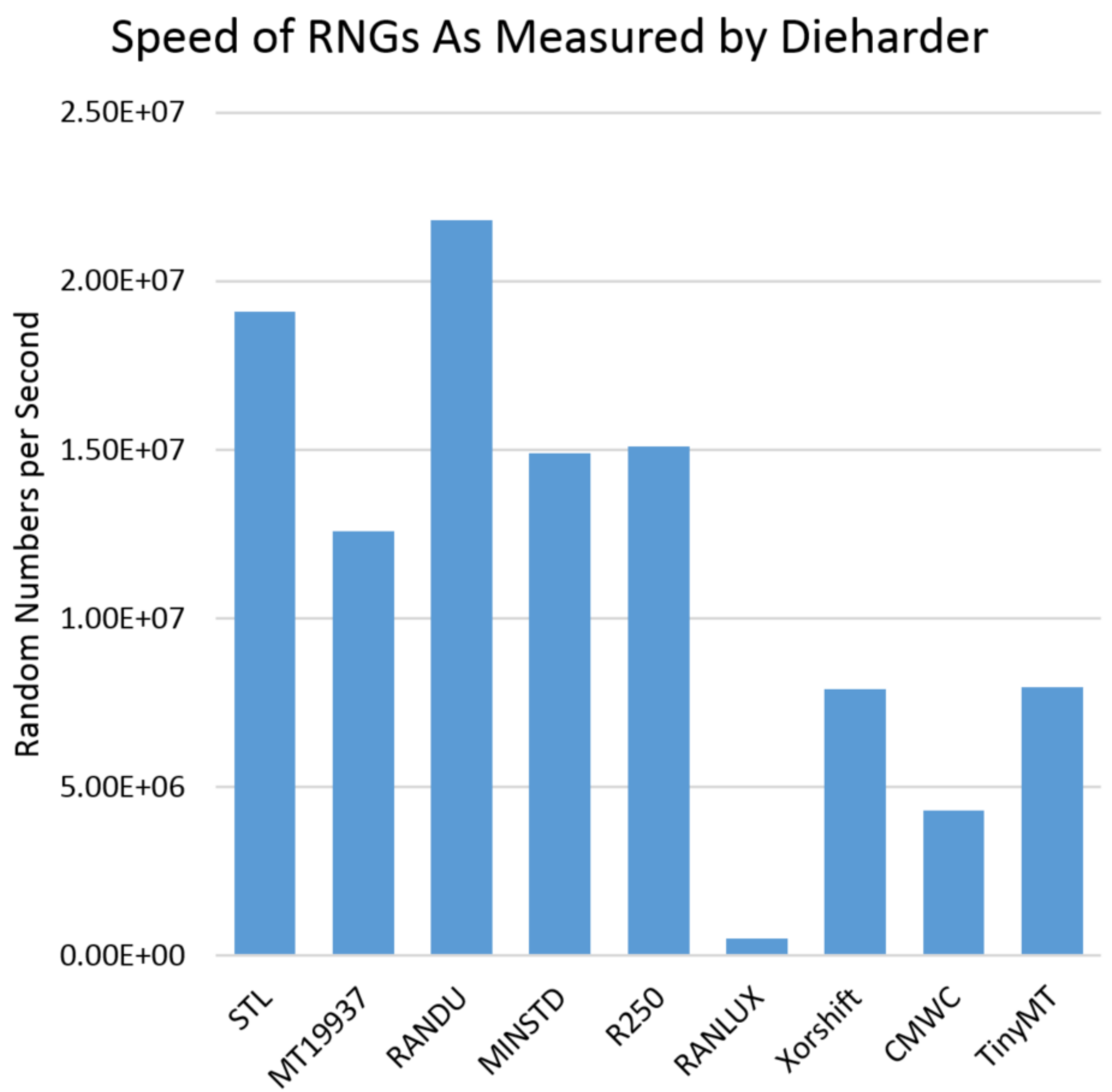
Dieharder Test Results

- RANLUX incomplete because running a full run would take ~100 hours.
- RANDU terrible.
- Everything else is about the same amount of good, with the best possible parameters.



Dieharder Speed Results

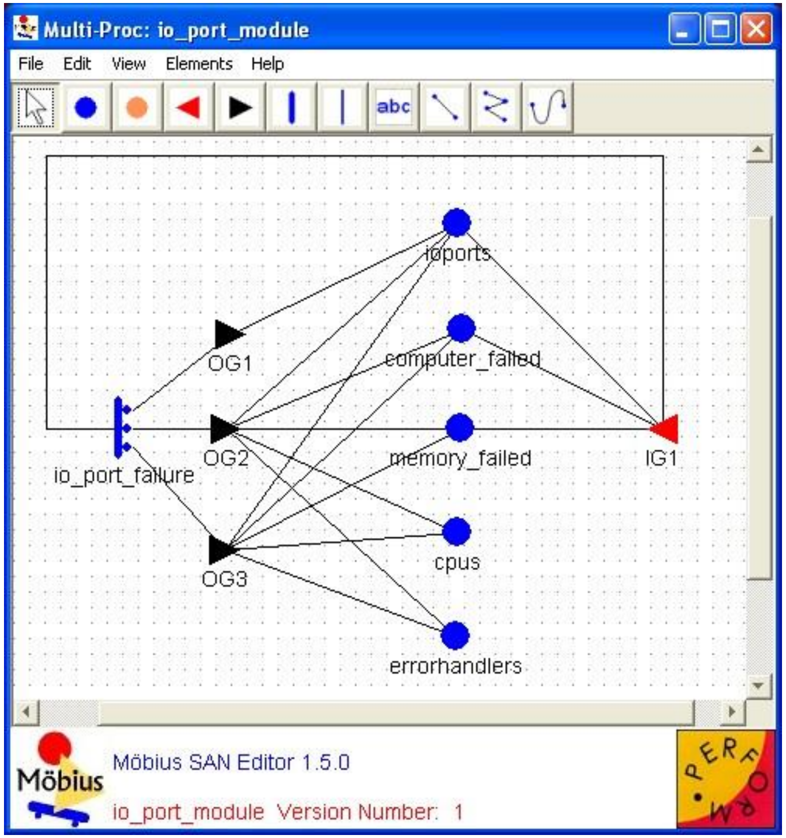
- RANLUX is extremely slow.
- RANDU is fast but it's unusable, remember?
- CMWC would be faster if we made a smaller version that might now pass as many tests.
- We couldn't make our MT as fast as the C++ STL. Go figure.



Conclusion

What Should I Use?

- Mersenne Twister is the common, easiest choice
  - …despite there being potentially faster generators out there
- New Generators being developed to address MT problems (have not seen adoption)
  - WELL
  - SFMT
- System Designer must decide!



Mobius, a simulator which uses RNGS

Thank You!

Questions?