TestU01: Empirical Statistical Testing of Random Number Generators

SmallCrush, Crush, and BigCrush Batteries

Prof. Michael Mascagni

Department of Computer Science
Department of Mathematics
Department of Scientific Computing
Graduate Program in Molecular Biophysics
Florida State University, Tallahassee, FL 32306 **USA**
E-mail: mascagni@fsu.edu
URL: http://www.cs.fsu.edu/∼mascagni

Outline

What is TestU01?

Definition

TestU01 is an open-source ANSI C library developed by **Pierre L'Ecuyer** and **Richard Simard** at the Université de Montréal (2007) for empirical statistical testing of random number generators (RNGs).

Key Features:

- Over 160 distinct statistical tests organized in modules
- Three predefined batteries of increasing severity
- Supports built-in and user-defined generators via a clean API
- Produces p-values for every test statistic
- Industry and academic gold standard for RNG validation

Primary Reference

L'Ecuyer & Simard, "TestU01: A C Library for Empirical Testing of Random Number Generators," *ACM Trans. Math. Softw.*, 33(4), 2007.

Why Test RNGs at All?

The Core Problem:

- Deterministic algorithms cannot produce truly random output
- Statistical flaws create exploitable patterns
- A generator that *looks* random may fail under rigorous scrutiny

Real-World Consequences of Weak RNGs:

- **Online Poker (2000s):** Card shuffles predictable from first few cards dealt
- **Netscape SSL (1995):** 40-bit keys seeded with time and PID; cracked in seconds
- **Monte Carlo Science:** Correlated sequences bias simulation results
- **Video Games:** Procedural generation exploitable by speed-runners

Key Insight

Passing a visual inspection or simple frequency test is not sufficient. Subtle correlations require hundreds of carefully designed statistical tests.

History and Context

Before TestU01

- Knuth tests (1969): spectral, serial, gap
- Diehard battery (Marsaglia, 1995): 18 tests
- NIST SP 800-22 (2001): 15 tests, crypto focus
- Fragmented tools, hard to extend

TestU01 (L'Ecuyer & Simard, 2007)

- Unified ANSI C library
- Modular: tests, batteries, generators separated
- 160+ tests; far more sensitive
- Detects flaws invisible to Diehard
- Used by NumPy, PCG, xoshiro designers

Notable Result

The Mersenne Twister (MT19937) — used as Python's default `random` module — **fails** multiple BigCrush tests. It is not cryptographically secure.

The Hypothesis Testing Framework

Setup for every TestU01 test:

- H_0: The sequence is produced by a perfect uniform RNG
- H_1: The sequence is *not* perfectly uniform or independent

Procedure:

1. Generate n values from the RNG under test
2. Compute a test statistic T from those values
3. Compute p-value $= P\big(T \geq t_{\text{observed}} \mid H_0\big)$
4. Flag if p-value ≈ 0 **or** p-value ≈ 1

TestU01 Default Decision Rule

- $p < 0.001$ **or** $p > 0.999 \implies$ test **FAILED**
- $0.001 \leq p \leq 0.999 \implies$ test **PASSED**

Understanding p-Values in This Context

Common misconceptions:

- A p-value is *not* the probability that the generator is truly random
- A single failure does not always prove a bad generator
- With 160 tests, roughly 0.16 spurious failures are expected by chance

How TestU01 handles multiple comparisons:

- Each test is run independently and reported separately
- Repeated failures across structurally different tests signal a real defect
- BigCrush (106 parameter settings): expect fewer than 1 spurious failure
- Results printed with both the p-value and a suspect flag ($*$)

Two-Sided Suspicion

TestU01 flags both $p < \varepsilon$ (too ordered/structured) and $p > 1 - \varepsilon$ (suspiciously uniform). A p-value of exactly 0.500 every single run is itself a warning sign.

Categories of Statistical Properties Tested

TestU01 tests cover six broad structural categories:

1. **Uniformity**: Are individual values uniform on $[0, 1)$?
2. **Independence**: Are successive values uncorrelated?
3. **Combinatorial**: Poker hands, coupons, birthday collisions
4. **Spectral**: FFT, linear complexity, lattice structure
5. **Graph/Walk**: Random walk statistics, GCD tests
6. **Compression**: Lempel-Ziv complexity, Hamming weights

Why Diversity Matters

A generator can pass every uniformity test and still fail combinatorial tests. True randomness requires *all* categories to be satisfied simultaneously.

Library Module Structure

TestU01 is organized into five logical groups:

Generator Modules

- `unif01` — abstraction layer (all generators conform to this)
- `ulcg`, `umrg` — built-in LCG and MRG generators
- `ucarry`, `ulfsr` — carry and LFSR generators
- `usoft` — wrappers for external generators

Test and Battery Modules

- `sres` — result data structures
- `sknuth` — Knuth classical tests
- `smarsa` — Marsaglia-style tests
- `snpair`, `svaria` — advanced tests
- `swalk` — random walk tests
- `bbattery` — predefined batteries

Design Philosophy

Tests and generators are fully decoupled. Any generator wrapped with `unif01_CreateExternGen01` can be run against any battery or any individual test with no changes to the test code.

The unif01_Gen Interface

Every generator in TestU01 exposes a uniform struct:

- `name` — human-readable identifier string
- `GetU01` — returns uniform double in $[0, 1)$
- `GetBits` — returns unsigned 32-bit integer
- `param` — opaque pointer to generator parameters
- `state` — opaque pointer to current state

Built-in generators include:

- LCG family (`ulcg`)
- Multiple Recursive (`umrg`)
- Lagged Fibonacci
- Combined generators (MRG32k3a)
- Mersenne Twister

Extending TestU01

User generators are wrapped with `unif01_CreateExternGen01`. This makes it trivial to test any RNG you write or encounter.

Wrapping a Custom Generator

```c
#include "unif01.h"
#include "bbattery.h"
#include <stdint.h>

/* A simple xorshift32 generator */
static uint32_t xstate = 123456789;

static double xorshift_U01 (void *par, void *sta) {
    xstate ^= xstate << 13;
    xstate ^= xstate >> 17;
    xstate ^= xstate << 5;
    return (double)xstate / 4294967296.0;
}
```

Wrapping a Custom Generator (Cont.)

```
static unsigned long xorshift_Bits (void *par, void *sta) {
    xstate ^= xstate << 13;
    xstate ^= xstate >> 17;
    xstate ^= xstate << 5;
    return (unsigned long)xstate;
}

int main (void) {
    unif01_Gen *gen;
    gen = unif01_CreateExternGen01 ("xorshift32", xorshift_U01);
    bbattery_SmallCrush (gen);
    unif01_DeleteExternGen01 (gen);
    return 0;
}
```

SmallCrush: Overview

Purpose

A quick screening battery for obvious weaknesses. Suitable for rapid initial evaluation and continuous integration feedback loops.

Scale

- **10 tests**, 15 statistics
- $\approx 5 \times 10^7$ values generated
- Runtime: **a few seconds**
- Minimal memory footprint

Scope

- Basic uniformity check
- Basic independence check
- Catches most weak generators
- Not sensitive to subtle flaws

Use Case

If a generator **fails SmallCrush**, it is clearly unsuitable for any serious use. Passing SmallCrush is a necessary but not sufficient condition for quality.

SmallCrush: The 10 Tests

#	Test Name	Module	Property Tested
1	BirthdaySpacings	smarsa	Spacing distribution in d-dimensional space
2	Collision	smarsa	Birthday paradox collision count
3	Gap	sknuth	Gaps between values in an interval
4	SimpPoker	sknuth	Poker hand frequencies
5	CouponCollector	sknuth	Coupon set completion lengths
6	MaxOft	sknuth	Maximum-of-t distribution
7	WeightDistrib	smarsa	Hamming weight distribution
8	MatrixRank	smarsa	GF(2) rank of binary matrices
9	HammingIndep	swalk	Hamming weight independence
10	RandomWalk1	swalk	Random walk (5 statistics)

SmallCrush: Test Details (Part 1)

Test 1 — BirthdaySpacings:

- Throw n points into m "days" (bins)
- Sort the points; compute spacings between consecutive values
- A true RNG produces Poisson-distributed collision counts
- **Catches LCGs and linear generators immediately**

Test 8 — MatrixRank:

- Fill an $r \times c$ binary matrix with consecutive output bits
- Compute the GF(2) rank; compare to the theoretical distribution
- Linear generators produce matrices with non-random rank profiles
- Low rank signals that rows are linearly dependent over GF(2)

SmallCrush: Test Details (Part 2)

Test 4 — SimpPoker:

- Draw groups of k values from $\{0, \ldots, d-1\}$
- Classify each group by its "poker hand" (all different, one pair, etc.)
- Compare frequencies to χ^2 theoretical distribution
- Sensitive to non-uniformity and short-range correlations

Test 10 — RandomWalk1 (5 statistics):

- Simulate a 1-D random walk from RNG output bits
- $+1$ for bit $= 1$, -1 for bit $= 0$
- Measures: H (last position), M (maximum), J (zero crossings), R (returns to origin), C (lag-1 correlation)
- Each statistic tests a different aspect of bit-level independence

Running SmallCrush

```c
#include "unif01.h"
#include "bbattery.h"

int main (void) {
    unif01_Gen *gen;

    /* Built-in Mersenne Twister seeded with 12345 */
    gen = unif01_CreateMT19937 (12345);

    /* Run the SmallCrush battery */
    bbattery_SmallCrush (gen);

    unif01_DeleteMT19937 (gen);
    return 0;
}
```

Compile with:

```
gcc -O2 -o test_mt mytest.c \
    -ltestu01 -lprobdist -lmylib -lm
```

Crush: Overview

Purpose

A thorough mid-level battery detecting moderate statistical weaknesses. The standard benchmark cited in academic RNG papers.

Scale

- **96 tests**, 144 statistics
- $\approx 2 \times 10^{10}$ values generated
- Runtime: **1–2 hours** on typical hardware
- Moderate memory usage

Scope

- Strict superset of SmallCrush
- Adds spectral and graph-based tests
- Includes string/compression tests
- Detects inter-dimensional correlations

Rule of Thumb

A generator passing Crush is considered **high quality** for simulation use. Most modern non-cryptographic PRNGs target passing this battery as a design goal.

Crush: Test Categories

Category	**Stat Count**	**Representative Tests**
Multinomial / Serial	18	SerialOver, MultinomialBitsOver
Birthday Spacings	10	BirthdaySpacings (dims 1–8)
Matrix Rank	8	BinaryRank (various $r \times c$)
Poker / Coupon	12	SimpPoker, CouponCollector
Run and Gap	12	Run, Gap, MaxOft
Spectral	16	Fourier3, LinearComp
Graph	8	Savir2, GCD
String / Compression	28	HammingCorr, LempelZiv
Walk / Misc	32	RandomWalk1, WeightDistrib

Crush: Three New Test Families

LinearComp (Berlekamp-Massey complexity):

- Computes the linear complexity L of a binary sequence of length n
- For a truly random sequence: $L \approx n/2$
- LFSR-based generators have **abnormally low** linear complexity
- Directly measures one key dimension of cryptographic weakness

Fourier3 (spectral / FFT test):

- Applies FFT to the output sequence mapped to ± 1
- Tests for periodicity or concentration of spectral energy
- Detects generators with short effective periods or resonances

LempelZiv:

- Compresses the bit sequence using the LZ78 algorithm
- Measures the number of distinct phrases (complexity proxy)
- A truly random sequence is incompressible; patterns reduce phrase count

Crush: Multidimensional Birthday Spacings

Birthday Spacings in d dimensions (Crush extension):

- Generate n points in a d-dimensional unit hypercube $[0,1)^d$
- Divide each axis into k equal intervals (giving k^d cells)
- Count collisions: two or more points landing in the same cell
- Under H_0: collision count follows a Poisson distribution with $\lambda = n^2/(2k^d)$

Why This Catches Linear Generators

An LCG with modulus m produces points in d-space that lie on at most $(d!\, m)^{1/d}$ parallel hyperplanes. Collision counts deviate sharply from Poisson — easily detected by dimensions $d \geq 3$.

BigCrush: Overview

Purpose

The most stringent battery in TestU01. Detects subtle statistical defects that survive Crush. The *de facto* gold standard for publication-quality RNG evaluation.

Scale

- **106 tests**, 160 statistics
- $\approx 4 \times 10^{10}$ values generated
- Runtime: **3–4 hours**
- Higher parameter settings than Crush

Scope

- Strict superset of Crush
- Same tests with larger n, higher d
- More sensitive to weak-bit correlations
- **MT19937 fails 2 tests**
- PCG64, xoshiro256** pass all 160

Important Caveat

Passing BigCrush is a strong claim of non-cryptographic quality. It does **not** imply cryptographic security.

BigCrush: Parameter Increases from Crush

Key sample-size increases (more statistical power):

Test	**Crush** n	**BigCrush** n
BirthdaySpacings	2^{20}	2^{21}
CollisionOver	2^{22}	2^{24}
MatrixRank	5 000	10 000
LinearComp	10^6	4×10^6
LempelZiv	2^{22}	2^{25}
HammingCorr	10^9	10^{10}
RandomWalk1	10^8	10^9

Larger n means greater statistical power: smaller deviations from uniformity become detectable.

BigCrush: Generator Scoreboard

Generator	BigCrush Failures	Notes
glibc `rand` (LCG)	Many	Fails most batteries
Mersenne Twister (MT19937)	2	BirthdaySpacings, MatrixRank
WELL19937c	0	Better equidistribution than MT
PCG64	0	Passes all; fast; permuted output
xoshiro256**	0	Passes all; very fast
xorshift64	1	Subtle matrix rank failure
ChaCha20 (CSPRNG)	0	Passes; cryptographically secure
RC4	Several	Deprecated; known bias

Takeaway: The MT19937 failures are not theoretical — they are reproducible and consistent. Use PCG64 or xoshiro256** for simulation work instead.

Installation

```
# Download and extract TestU01
wget http://simul.iro.umontreal.ca/testu01/TestU01.zip
unzip TestU01.zip && cd TestU01-1.2.3

# Configure, build, and install
./configure --prefix=/usr/local
make && sudo make install

# Compile your test program (link all three libs + math)
gcc -O2 -o mytest mytest.c \
    -ltestu01 -lprobdist -lmylib -lm
```

Required headers:

- `unif01.h` — generator abstraction layer
- `bbattery.h` — predefined battery entry points
- `smarsa.h`, `sknuth.h` — individual test modules
- `sres.h` — result data structures

Running the Three Batteries

```c
#include "unif01.h"
#include "bbattery.h"

int main (void) {
    unif01_Gen *gen;

    /* Use the built-in Mersenne Twister */
    gen = unif01_CreateMT19937 (12345);

    /* --- Choose ONE battery per run --- */
```

Running the Three Batteries (Cont.)

```
1
2       /* Quick screen: ~seconds */
3       bbattery_SmallCrush (gen);
4
5       /* Standard:  ~1-2 hours  (comment out SmallCrush first) */
6       /* bbattery_Crush (gen); */
7
8       /* Gold standard: ~3-4 hours */
9       /* bbattery_BigCrush (gen); */
10
11      unif01_DeleteMT19937 (gen);
12      return 0;
13  }
```

Run each battery as a separate program execution. Do not chain all three in one run — the generator state carries over.

Running an Individual Test

```
1  #include "unif01.h"
2  #include "smarsa.h"
3  #include "sres.h"
4  #include <stdio.h>
5
6  int main (void) {
7      unif01_Gen  *gen;
8      smarsa_Res  *res;
9
10     gen = unif01_CreateMT19937 (12345);
11     res = smarsa_CreateRes ();
```

Running an Individual Test (Cont.)

```
    /* BirthdaySpacings:
       N=1 replication, n=1000 samples, r=0 bits skipped,
       d=2^30 bins, t=2 dimensions, p=1 */
    smarsa_BirthdaySpacings (gen, res,
        1, 1000, 0, 1073741824, 2, 1);

    printf ("p-value = %f\n", res->Pois->pVal2[0]);

    smarsa_DeleteRes (res);
    unif01_DeleteMT19937 (gen);
    return 0;
}
```

Sample SmallCrush Output

```
=========================================
HOST = myhost,  Linux
xorshift32
=========================================

SmallCrush
-----------------------------------------------
Test                          p-value
-----------------------------------------------
 1  BirthdaySpacings          9.9e-04  *
 2  Collision                 0.4821
 3  Gap                       0.2371
 4  SimpPoker                 0.8102
 5  CouponCollector           0.3147
 6  MaxOft                    0.5563
 7  WeightDistrib             0.4219
 8  MatrixRank                2.0e-04  *
 9  HammingIndep              0.2187
10  RandomWalk1 H             0.4823
10  RandomWalk1 M             0.9631
10  RandomWalk1 J             0.3415
10  RandomWalk1 R             0.2951
10  RandomWalk1 C             0.6078
-----------------------------------------------
All tests were done.
Number of failures: 2
```

Lines marked $*$ indicate $p < 0.001$ or $p > 0.999$ — both are **failures**.

Interpreting Failures

Step 1 — Count failures:

- 0 failures: **generator passes this battery**
- 1–2 failures: **borderline — re-run with at least 3 different seeds**
- 3+ failures: **generator has a real structural defect**

Step 2 — Identify which tests fail (diagnose the cause):

- BirthdaySpacings / MatrixRank failure $\Rightarrow$ **linear structure**
- LinearComp failure $\Rightarrow$ **low algebraic complexity** (LFSR-like)
- LempelZiv failure $\Rightarrow$ **compressible output** (repeating patterns)
- RandomWalk failure $\Rightarrow$ **bit-level autocorrelation**
- Fourier3 failure $\Rightarrow$ **spectral periodicity** in the output

Step 3 — Test the bit-reversed stream:

- Wrap with `unif01_CreateBitReverse`
- Some generators pass forward but fail reversed — another structural flaw

TestU01 vs Other Test Suites

Suite	Stats	Sensitivity	Time	Use Case
Diehard (Marsaglia)	18	Low–Med	Seconds	Legacy comparison
NIST SP 800-22	15	Medium	Seconds	Crypto certification
PractRand	Many	High	Variable	Stream/online testing
TestU01 SmallCrush	15	Medium	Seconds	Rapid screening
TestU01 Crush	144	High	1–2 hrs	Academic standard
TestU01 BigCrush	160	Very High	3–4 hrs	Publication gold std

Recommendation:

- Use **BigCrush** for any RNG intended for research or simulation
- Use **NIST SP 800-22** additionally for cryptographic generators
- **Do not rely on Diehard alone** — it misses many modern flaws

Five Common Pitfalls

1. **Testing only one seed.** Some generators fail only with specific initial states. Always test with at least three to five independent seeds.
2. **Ignoring $p > 0.999$ failures.** "Too uniform" is just as suspicious as "too structured." Suspiciously perfect p-values indicate the output is being artificially regularized.
3. **Confusing BigCrush with cryptographic security.** Statistical indistinguishability is not the same as computational indistinguishability. Use NIST SP 800-90A (CTR_DRBG) for security work.
4. **Testing the wrong bit slice.** Test the exact bits you actually use. Some generators fail when only the lower 32 bits are extracted and the upper bits are discarded.
5. **Stopping after SmallCrush.** SmallCrush misses many real defects that Crush or BigCrush reveal. Always run at least Crush for any production-quality simulation.

Choosing the Right Battery

Situation	Battery	Time	Confidence
Initial screening / CI	SmallCrush	Seconds	Low
Development testing	SmallCrush	Seconds	Low
Pre-release QA	Crush	1–2 hrs	High
Research publication	BigCrush	3–4 hrs	Very High
Cryptographic RNG eval	BigCrush + NIST	4+ hrs	Gold Std

Decision Rules:

- Fails SmallCrush $\Rightarrow$ discard immediately
- Passes SmallCrush, fails Crush $\Rightarrow$ unsuitable for serious simulation
- Passes Crush, fails BigCrush $\Rightarrow$ borderline; avoid for research
- Passes BigCrush $\Rightarrow$ **suitable for non-cryptographic use**

Key Takeaways

1. TestU01 is the **gold standard** for empirical RNG testing, with 160+ statistics organized in three batteries of increasing rigor.
2. **SmallCrush** (15 stats, seconds) screens for obvious flaws. **Crush** (144 stats, hours) is the academic standard. **BigCrush** (160 stats, hours) is the most stringent available.
3. p-values outside [0.001, 0.999] indicate failure in *either* direction. Multiple failures across diverse tests confirm a real structural defect.
4. MT19937 (Python's `random`) **fails BigCrush**. Use **PCG64** or **xoshiro256**** for simulation.
5. Passing BigCrush does **not** imply cryptographic security. Use NIST SP 800-90A (CTR_DRBG, Hash_DRBG) for security applications.
6. Always test with multiple seeds and in the bit-reversed orientation.

Further Reading

- L'Ecuyer & Simard (2007). "TestU01: A C Library for Empirical Testing of Random Number Generators." *ACM Trans. Math. Softw.*, 33(4), Article 22.
- Knuth, D.E. (1997). *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed. Addison-Wesley.
- O'Neill, M.E. (2014). "PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation." HMC Technical Report HMC-CS-2014-0905.
- Blackman & Vigna (2021). "Scrambled Linear Pseudorandom Number Generators." *ACM Trans. Math. Softw.*, 47(4).
- NIST SP 800-90A Rev. 1 (2015). "Recommendation for Random Number Generation Using Deterministic Random Bit Generators."

- **TestU01 homepage:** `http://simul.iro.umontreal.ca/testu01/tu01.html`
- **PCG generator:** `https://www.pcg-random.org`
- **xoshiro / xoroshiro:** `https://prng.di.unimi.it`

Questions?

TestU01

Empirical Statistical Testing of Random Number Generators

Useful Links

- TestU01 source code: `simul.iro.umontreal.ca/testu01`
- PCG generator family: `pcg-random.org`
- xoshiro / xoroshiro: `prng.di.unimi.it`
- PractRand (complementary suite): `pracrand.sourceforge.net`

Questions?