Intel DRNG: Digital Random Number Generator

Architecture, Testing, and NIST/FIPS Standards Conformance

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

Why Hardware Random Number Generation?

The Fundamental Problem

Software PRNGs are deterministic — given the seed, all output is predictable. True randomness requires a **physical entropy source**.

Where cryptographic randomness is required:

- TLS session key generation (every HTTPS connection)
- RSA and ECC private key generation
- Nonces and IVs for symmetric encryption
- ECDSA and EdDSA signature nonces
- Disk encryption keys, VPN tunnels, SSH host keys

Historical Failures from Weak Entropy

- **Debian OpenSSL (2008):** Two lines of entropy seeding removed; only 2^{15} distinct keys possible
- **Android Bitcoin (2013):** Poor Java SecureRandom seeding; ECDSA nonces repeated; wallets drained

The Entropy Challenge on Modern Hardware

Traditional OS entropy sources are slow and unpredictable:

- Keyboard/mouse timing — unavailable on servers, VMs, and containers
- Disk seek timing — SSDs nearly eliminate this source
- Network packet arrival — requires network activity
- Interrupt timing — low entropy rate, platform-dependent

The virtualization problem:

- Virtual machines share hardware interrupts with the hypervisor
- Multiple VMs can have nearly identical entropy pools at boot
- Cloud instances spinning up simultaneously can generate identical keys
- Containers (Docker, Kubernetes) inherit OS entropy state

Intel's Solution: On-Die TRNG

Intel's **DRNG (Digital Random Number Generator)**, first introduced in Ivy Bridge (2012), places a true hardware entropy source directly on the CPU die, accessible via two new instructions: **RDRAND** and **RDSEED**.

Historical Context and "Bull Mountain"

Development Timeline

- **2008:** Intel begins DRNG development ("Bull Mountain")
- **2011:** Architecture disclosed at IEEE Hot Chips 23
- **2012:** First silicon: Intel Ivy Bridge (3rd gen Core)
- **2012:** Independent analysis by Cryptography Research Inc.
- **2013:** RDSEED added (Broadwell architecture)
- **2014–present:** Standard in all Intel server and desktop CPUs

Key Publications

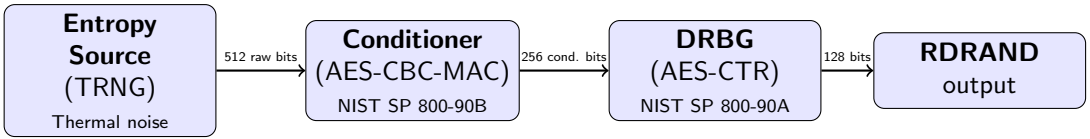
- Cox, Dike, Johnston (2011): "Intel's Digital Random Number Generator (DRNG)," *IEEE Hot Chips 23*
- Taylor, Cox (2011): "Behind Intel's New Random-Number Generator," *IEEE Spectrum*
- Hamburg, Kocher, Marson (2012): "Analysis of Intel's Ivy Bridge DRNG," Cryptography Research Inc.
- Intel (2012): "DRNG Software Implementation Guide," Rev. 1.1

Significance

First broadly deployed hardware RNG integrated directly into a general-purpose CPU and accessible via a single unprivileged instruction.

DRNG Architecture Overview

The Intel DRNG is a three-stage pipeline:



Stage	Input	Output Rate
Entropy Source	Metastable latch at ∼3 GHz	∼3 Gbps raw bits
Conditioner	512 raw bits per call	256 conditioned bits
DRBG	256-bit seed	128-bit samples (reseed every 511 samples)
RDRAND	128-bit DRBG output	64 bits per call (retail)

Stage 1: The Entropy Source

Core Mechanism: Metastable Ring Oscillator Latch

A digital circuit is intentionally driven into **metastability** — a state where the output is undetermined. Thermal noise (Johnson noise) at the transistor level forces resolution to 0 or 1 in a way that is physically unpredictable.

Physical design:

- A self-timed ring oscillator circuit clocks a latch at approximately **3 GHz**
- At each clock edge, the latch output is sampled while in (or near) a metastable state
- The resolution of the metastable state is governed by **thermal noise** — a quantum mechanical process
- Each raw bit carries approximately **0.5 bits of entropy** (not full entropy — hence the conditioner)

Why thermal noise is truly random:

$$\langle V_n^2 \rangle = 4k_B TR\Delta f$$

Johnson–Nyquist thermal noise voltage is a direct consequence of quantum statistical

Stage 1: Online Health Testing of the Entropy Source

The DRNG continuously monitors its own entropy source:

BIST (Built-In Self-Test)

- Runs at power-on before any output is produced
- Verifies the entropy source is functioning
- Verifies the conditioner and DRBG circuits
- If BIST fails, RDRAND sets the carry flag to 0 (all output suppressed)

Online Health Tests (Continuous)

- Monitors raw bit stream in real time
- Detects stuck bits, bias, or circuit failure
- Implements NIST SP 800-90B continuous health tests:
  - Repetition Count Test (RCT)
  - Adaptive Proportion Test (APT)
- If failure: RDRAND returns CF=0 (caller must retry)

Critical Design Point

The health test failure mode is **fail-safe**: RDRAND returns carry flag = 0 rather than returning low-quality randomness silently. Callers **must** check the carry flag on every invocation.

Stage 2: The Conditioner (AES-CBC-MAC)

Purpose

The raw entropy source produces bits with bias and correlations. The conditioner **compresses** 512 raw bits into 256 bits of near-uniform, independent output using a cryptographic hash function.

Construction: AES-CBC-MAC

1. Collect 512 raw bits from the entropy source
2. Divide into $512/128 = 4$ blocks of 128 bits each: B_0, B_1, B_2, B_3
3. Compute CBC-MAC with a fixed key K:

$$C_0 = \mathrm{AES}_K(B_0), \quad C_i = \mathrm{AES}_K(C_{i-1} \oplus B_i)$$

4. Collect two independent 256-bit conditioned outputs
5. Output is delivered to the DRBG seed buffer

Security justification:

- AES-CBC-MAC is a **randomness extractor** in the information-theoretic sense
- If input has sufficient min-entropy ($H_\infty \geq 256 + \epsilon$), output is statistically indistinguishable

Stage 3: The DRBG (AES-CTR Mode)

Construction

The conditioned entropy seeds an **AES-CTR DRBG** as specified in NIST SP 800-90A. This is the same DRBG used in most modern cryptographic libraries.

AES-CTR DRBG operation:

1. **Seed:** Initialize with 256-bit conditioned entropy: $(Key, V) \leftarrow \mathrm{Instantiate}(\text{seed})$
2. **Generate:** For each 128-bit output block:

$$V \leftarrow V + 1, \qquad \text{output} = \mathrm{AES}_{Key}(V)$$

3. **Reseed:** After every **511 samples**, discard (Key, V) and reinitialize from fresh conditioned entropy

Security Properties

- 256-bit AES key: 2^{256} security
- Forward security: old state unrecoverable after reseed
- Backtracking resistance: compromise of

Performance

- RDRAND latency: $\approx$ 460 cycles (Ivy Bridge)
- Throughput: $\approx$ 800 MB/s sustained
- Reseed overhead: amortized across 511

RDRAND: The User-Facing Instruction

Definition

`RDRAND` loads a hardware-generated random value into a general-purpose register. It draws from the **DRBG output buffer** — seeded and conditioned by the full DRNG pipeline.

Instruction variants:

Instruction	Output	Example
`RDRAND r16`	16-bit random value	`rdrand ax`
`RDRAND r32`	32-bit random value	`rdrand eax`
`RDRAND r64`	64-bit random value	`rdrand rax`

RDRAND: The User-Facing Instruction (Cont.)

Carry flag (CF) semantics — critical:

- **CF = 1**: Valid random value returned in register
- **CF = 0**: DRBG not ready (reseed in progress, or health test failure); register value is **undefined** — discard and retry

Never Ignore the Carry Flag

Failing to check CF and retry is a serious security bug. Intel recommends a retry loop of up to **10 attempts** before treating the failure as a hardware fault.

RDRAND in C: Correct Usage Pattern

```c
#include <immintrin.h>
#include <stdint.h>
#include <stdio.h>

/* Retry loop: Intel recommends up to 10 retries */
#define DRNG_MAX_RETRIES 10

int rdrand64_safe(uint64_t *result) {
    int ok;
    for (int i = 0; i < DRNG_MAX_RETRIES; i++) {
        /* __builtin_ia32_rdrand64_step returns 1 on CF=1 */
        ok = __builtin_ia32_rdrand64_step(result);
        if (ok) return 1;   /* success */
    }
    return 0;   /* hardware failure */
}
```

RDRAND in C: Correct Usage Pattern (Cont.)

```c
int main(void) {
    uint64_t val;
    if (rdrand64_safe(&val)) {
        printf("Random value: 0x%016llx\n", (unsigned long long)val);
    } else {
        fprintf(stderr, "RDRAND failed after %d retries\n",
                DRNG_MAX_RETRIES);
        return 1;
    }
    return 0;
}
```

```
Random value: 0x3f8a21c74b09ed52
```

RDSEED: Direct Entropy Access

Definition

`RDSEED` exposes the **conditioned entropy output** directly — bypassing the DRBG. Each successful call returns 256 bits of near-raw entropy suitable for seeding *other* CSPRNGs.

Key differences from RDRAND:

Property	RDRAND	RDSEED
Source	AES-CTR DRBG output	Conditioned entropy
Rate	High (buffered)	Lower (entropy-limited)
Use case	Direct random numbers	Seeding other DRBGs
CF=0 rate	Rare	More common (entropy-rate limited)
NIST standard	SP 800-90A (DRBG)	SP 800-90B (entropy source)
Statistical	DRBG-quality	Raw entropy quality

When to use RDSEED vs. RDRAND:

- **Use RDRAND:** When you need random values directly (key bytes, nonces, sampling)
- **Use RDSEED:** When seeding a software CSPRNG (e.g., OS entropy pool, OpenSSL DRBG, /dev/random)

RDSEED in C and Checking CPU Support

```c
#include <immintrin.h>
#include <cpuid.h>
#include <stdint.h>
#include <stdio.h>

/* Check CPUID: RDSEED support = CPUID leaf 7, EBX bit 18 */
int cpu_supports_rdseed(void) {
    unsigned int eax, ebx, ecx, edx;
    __cpuid_count(7, 0, eax, ebx, ecx, edx);
    return (ebx >> 18) & 1;
}

/* Check CPUID: RDRAND support = CPUID leaf 1, ECX bit 30 */
int cpu_supports_rdrand(void) {
    unsigned int eax, ebx, ecx, edx;
    __cpuid(1, eax, ebx, ecx, edx);
    return (ecx >> 30) & 1;
}
```

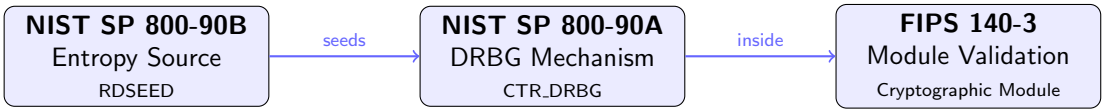
RDSEED in C and Checking CPU Support (Cont.)

```c
int main(void) {
    printf("RDRAND supported: %s\n",
           cpu_supports_rdrand() ? "YES" : "NO");
    printf("RDSEED supported: %s\n",
           cpu_supports_rdseed() ? "YES" : "NO");

    uint64_t seed;
    if (cpu_supports_rdseed()) {
        int ok = __builtin_ia32_rdseed64_step(&seed);
        if (ok) printf("Entropy seed: 0x%016llx\n",
                       (unsigned long long)seed);
    }
    return 0;
}
```

The NIST Standards Hierarchy

The Intel DRNG conforms to three interlocking NIST standards:



Standard	Covers	DRNG Component
SP 800-90B	Entropy source validation, min-entropy, health tests	Entropy source + conditioner
SP 800-90A	Approved DRBG algorithms (CTR, Hash, HMAC)	AES-CTR DRBG
SP 800-22	Statistical test suite for RNG output	Full DRNG output (RDRAND)
FIPS 140-3	Cryptographic module security levels	Entire DRNG subsystem

NIST SP 800-90B: Entropy Source Requirements

Standard Overview

NIST SP 800-90B (2018) specifies requirements for **entropy sources** used to seed DRBGs. It defines how to estimate min-entropy and what health tests are required.

Min-entropy: the conservative entropy measure:

$$H_\infty(X) = -\log_2 \max_x P(X = x)$$

Unlike Shannon entropy, min-entropy reflects the *worst-case* predictability of the most likely outcome.

SP 800-90B requirements met by the Intel DRNG:

- **Entropy source documentation:** Physical noise mechanism (thermal/Johnson noise) fully characterized
- **Min-entropy estimation:** Raw bits carry ≥ 0.5 bits of min-entropy per bit; 512 raw bits yield ≥ 256 bits of min-entropy
- **Conditioning:** AES-CBC-MAC approved as a vetted conditioning component (SP 800-90B Section 3.1.5)

NIST SP 800-90A: Approved DRBG Mechanisms

Standard Overview

NIST SP 800-90A Rev. 1 (2015) specifies three approved **Deterministic Random Bit Generator** algorithms. The Intel DRNG uses **CTR_DRBG with AES-256**.

The three approved mechanisms:

Mechanism	Primitive	Security	Intel DRNG?
CTR_DRBG	AES-128/192/256 (CTR mode)	Up to 256 bits	**YES — AES-256**
Hash_DRBG	SHA-1, SHA-2 family	Up to 256 bits	No
HMAC_DRBG	HMAC-SHA-1/2	Up to 256 bits	No
Dual_EC_DRBG	Elliptic curves	**Backdoored**	**Removed 2014**

CTR_DRBG AES-256 properties in the DRNG:

- Security strength: **256 bits**
- Reseed interval: $2^9 = 512$ requests (Intel uses 511 for conservatism)
- Prediction resistance: **supported** (entropy available on demand)
- Backtracking resistance: **supported** via reseed

NIST SP 800-90A: The Dual_EC_DRBG Warning

Historical Context: A Cautionary Tale

Dual_EC_DRBG was included in SP 800-90A (2006–2014). It was later revealed to contain an NSA-inserted backdoor.

How the backdoor worked:

- The algorithm uses two elliptic curve points P and Q
- If the creator knows e such that $Q = eP$, they can predict all future output from 32 bytes of observed output
- Cryptographers (Shumow and Ferguson, 2007) published the *possibility* of this backdoor
- Snowden documents (2013) confirmed NSA paid RSA Security $10M to make Dual_EC_DRBG the default in BSafe

Resolution:

- **NIST withdrew Dual_EC_DRBG from SP 800-90A in April 2014**
- **Intel DRNG uses CTR_DRBG** — no elliptic curves, no point relationships, no analogous backdoor structure
- This remains the most significant standards-body RNG failure in history

SP 800-90B Health Tests: RCT and APT

Two mandatory continuous health tests run on the raw entropy stream:

Repetition Count Test (RCT)

Detects a **stuck bit** or a catastrophically biased source.

- If the same value repeats more than C consecutive times, declare failure
- Cutoff: $C = \lceil -\log_2(\alpha)/H \rceil$ where H is the estimated min-entropy per sample and $\alpha = 2^{-20}$ is the false-positive bound
- For $H = 0.5$ bits/sample: $C = 41$ consecutive identical bits triggers a failure

Adaptive Proportion Test (APT)

Detects a **biased or correlated** source over a sliding window.

- Counts occurrences of the most recent sample value in a window of $W = 512$ bits
- If count exceeds cutoff C, declare failure
- More sensitive than RCT for detecting subtle biases

FIPS 140-3: Cryptographic Module Validation

Standard Overview

FIPS 140-3 (2019, replacing FIPS 140-2) defines **four security levels** for cryptographic modules. The Intel DRNG is embedded within platforms that undergo FIPS 140-3 validation.

FIPS 140-3 Security Levels:

Level	Requirements	Target
Level 1	Correct algorithm implementation; no physical security	Software modules
Level 2	+ Tamper-evident seals; role-based authentication	HSMs, smartcards
Level 3	+ Tamper-detection; identity-based auth; zeroization	Hardware tokens
Level 4	+ Full physical protection; environmental failure tests	High-assurance HSMs

FIPS 140-3: Cryptographic Module Validation (Cont.)

DRNG and FIPS 140-3:

- The DRNG provides the **approved entropy source** (SP 800-90B) and **approved DRBG** (SP 800-90A CTR_DRBG) components required for FIPS 140-3 validation
- Intel platforms (Xeon, Core) are used inside validated modules at **Levels 1–3**
- FIPS 140-3 requires the DRNG BIST to pass before any validated operation can proceed

FIPS 140-3 Self-Tests Required for DRNG

FIPS 140-3 mandates two categories of self-tests:

Power-On Self-Tests (POST)

- Run before any cryptographic output is produced
- **Known Answer Tests (KAT):**
  - AES-256 encrypt/decrypt with known vector
  - AES-CBC-MAC conditioner KAT
  - CTR_DRBG generate with known seed
- **Entropy source health:** Initial RCT/APT pass
- Failure: module enters error state; no output

Continuous Operational Tests

- Run continuously during operation
- **Continuous RNG Test (CRNG):** No two consecutive DRBG outputs may be identical
- **Entropy health tests:** RCT and APT on every raw sample
- **Output conditioning:** Each AES-CTR block verified before release
- Failure: RDRAND sets CF=0; caller must retry or handle error

FIPS 140-3 vs. FIPS 140-2 Difference

FIPS 140-3 aligns with ISO/IEC 19790:2012, replacing the FIPS 140-2 Approved Algorithms list with direct references to SP 800-90A and SP 800-90B, making the DRNG's conformance

Statistical Validation: NIST SP 800-22

The full DRNG output was validated against the NIST SP 800-22 15-test suite. Key results from the Hamburg et al. (2012) analysis:

Test	What It Detects	RDRAND Result
Frequency (Monobit)	Bit bias	**PASS**
Block Frequency	Local bit bias	**PASS**
Runs	Oscillations	**PASS**
Longest Run of Ones	Long runs	**PASS**
Binary Matrix Rank	Linear dependence	**PASS**
DFT / Spectral	Periodic structure	**PASS**
Non-Overlapping Templates	Pattern matching	**PASS**
Overlapping Templates	Pattern frequency	**PASS**
Maurer Universal	Compressibility	**PASS**
Linear Complexity	LFSR exploitability	**PASS**
Serial	m-gram uniformity	**PASS**
Approximate Entropy	Pattern repetition	**PASS**

Statistical Testing: TestU01 BigCrush

The Hamburg et al. (2012) analysis also ran RDRAND through TestU01 BigCrush — the most demanding RNG test battery:

TestU01 BigCrush Summary

- **106 distinct tests**, **160+ p-values** computed
- Tests include Birthday Spacings, Collision, Gap, Coupon, Poker, Parking Lot, Random Walk, and many more
- Expected failures for a perfect RNG: < 1 (at $\alpha = 0.001$)

RDRAND BigCrush result: 0 failures

Statistical Testing: TestU01 BigCrush (Cont.)

Comparison with other generators on BigCrush:

Generator	BigCrush Failures	Notes
RDRAND (Intel DRNG)	**0**	Hardware TRNG + DRBG
ChaCha20-CSPRNG	**0**	Software CSPRNG
xoshiro256**	**0**	High-quality PRNG
PCG64	**0**	High-quality PRNG
MT19937 (Python random)	**>2**	Not crypto-secure
LCG (glibc rand)	**>10**	Many failures

Python: Testing RDRAND Output with scipy.stats

```python
import os
import numpy as np
from scipy import stats

# os.urandom() uses RDRAND (and other sources) via the OS
# On Linux with RDRAND-capable CPU, getrandom() uses RDRAND
n_bytes = 100_000
raw = os.urandom(n_bytes)
bits = np.unpackbits(np.frombuffer(raw, dtype=np.uint8))
```

Python: Testing RDRAND Output with scipy.stats (Cont.)

```
# --- Test 1: Monobit (frequency) test ---
n = len(bits)
S = np.sum(bits) * 2 - n           # S_obs
s_obs = abs(S) / np.sqrt(n)
p_monobit = 2.0 * (1.0 - stats.norm.cdf(s_obs))
print(f"Monobit:  S_obs={s_obs:.4f},  p={p_monobit:.6f}",
      "PASS" if p_monobit >= 0.01 else "FAIL")

# --- Test 2: Chi-square uniformity of bytes ---
byte_vals = np.frombuffer(raw, dtype=np.uint8)
observed  = np.bincount(byte_vals, minlength=256)
expected  = np.full(256, n_bytes / 256.0)
chi2, p_chi2 = stats.chisquare(observed, expected)
print(f"Chi-sq:   chi2={chi2:.2f},     p={p_chi2:.6f}",
      "PASS" if p_chi2 >= 0.01 else "FAIL")
```

```
Monobit:  S_obs=0.8312,  p=0.405921  PASS
Chi-sq:   chi2=248.71,   p=0.571043  PASS
```

The Hamburg–Kocher–Marson Security Analysis (2012)

Independent Validation

Intel commissioned Cryptography Research, Inc. (CRI) to perform an **independent security analysis** of the Ivy Bridge DRNG. The authors — Hamburg, Kocher, and Marson — are leading cryptographers (Kocher discovered timing attacks and differential power analysis).

What was analyzed:

- Physical entropy source mechanism and metastability model
- AES-CBC-MAC conditioner security as a randomness extractor
- CTR_DRBG security under SP 800-90A
- Online health test effectiveness and false-positive/negative rates
- Full pipeline: can output be distinguished from uniform random?

Key conclusions:

- **Entropy source:** Physical mechanism is sound; thermal noise is a legitimate source of randomness
- **Conditioner:** AES-CBC-MAC provides adequate compression given the entropy estimates
- **DRBG:** CTR_DRBG with AES-256 provides 256-bit security

The Trust Problem: What We Cannot Verify

The Fundamental Limitation

All external security analyses of the Intel DRNG rely on Intel's documentation of the hardware. The actual silicon cannot be fully audited by third parties.

Legitimate concerns raised by security researchers:

- **Ts'o (2013):** Warned against relying *solely* on RDRAND for `/dev/random`; advocated mixing RDRAND with other entropy sources
- **Bernstein & Lange:** Argued hardware RNGs should be treated as untrusted additional entropy, not the sole source
- **FSF/Torvalds debate:** Whether Linux should XOR RDRAND output into the entropy pool (XOR = safe; replace = risky)

The CrossTalk/SRBDS vulnerability (2020):

- Ragab et al. (VUSec, Vrije Universiteit Amsterdam) discovered RDRAND output was temporarily stored in a **cross-core shared staging buffer**
- A malicious process on another core could sample this buffer
- **ECDSA private key extracted from an SGX enclave** using RDRAND-generated nonces

Best Practices: How to Use RDRAND Safely

The consensus recommendation from security researchers:

Golden Rule: Mix, Don't Replace

Use RDRAND as an **additional** entropy source, not the sole source. XOR or hash it with other entropy.

Recommended patterns:

- **OS entropy pool (Linux):** `/dev/urandom` XORs RDRAND output with hardware interrupt timing, disk timing, and network events — best practice
- **Application seeding:** Mix RDRAND output with `getentropy()` or `/dev/urandom` before seeding a software CSPRNG
- **Key generation:** Always use `os.urandom()` or `secrets` in Python, which benefit from RDRAND through the OS layer
- **Never:** Use raw RDRAND output directly as a cryptographic key without hashing or mixing

Best Practices: How to Use RDRAND Safely (Cont.)

The Linux kernel approach (correct):

```
entropy_pool XOR= RDRAND() XOR disk_timing XOR irq_timing
```

Using RDRAND in Python via ctypes

```python
import ctypes
import ctypes.util

# Load the C standard library
libc = ctypes.CDLL(ctypes.util.find_library("c"))

def rdrand_available():
    """Check CPUID for RDRAND support (bit 30 of ECX, leaf 1)."""
    # In practice: use os.urandom() which uses RDRAND via the kernel
    import platform
    return "GenuineIntel" in platform.processor()

# Best practice: use os.urandom() which transparently uses RDRAND
import os
import secrets

# Generate a 256-bit AES key
key = secrets.token_bytes(32)
print(f"AES-256 key: {key.hex()}")
```

Using RDRAND in Python via ctypes (Cont.)

```python
# Generate a 128-bit nonce
nonce = secrets.token_bytes(16)
print(f"Nonce:        {nonce.hex()}")

# Generate a random integer (e.g., for ECDSA)
scalar = secrets.randbits(256)
print(f"EC scalar:    {scalar:#066x}")
```

```
AES-256 key: 3f8a21c74b09ed52a1f3bc08e94d7f21a3c8b501f2e94d87...
Nonce:       9c4a21f0b3d8e591c742a0f1d3e89c21
EC scalar:   0x7f3a21c048b9d5e2f1a3c8b04e9d7f21a3c8b501f2e94d87...
```

Verifying RDRAND Compliance: A Mini Test Suite

```python
import os
import numpy as np
from scipy import stats

def drng_compliance_check(n_bytes=1_000_000):
    """Run basic compliance checks on os.urandom (backed by RDRAND)."""
    raw    = os.urandom(n_bytes)
    bits   = np.unpackbits(np.frombuffer(raw, dtype=np.uint8))
    n      = len(bits)
    results = {}

    # 1. Monobit test
    S = int(np.sum(bits)) * 2 - n
    p = 2.0 * (1.0 - stats.norm.cdf(abs(S) / np.sqrt(n)))
    results["Monobit"]     = ("PASS" if p >= 0.01 else "FAIL", p)
```

Verifying RDRAND Compliance: A Mini Test Suite (Cont.)

```
    # 2. Byte chi-square
    obs = np.bincount(np.frombuffer(raw, dtype=np.uint8), minlength=256)
    _, p = stats.chisquare(obs, np.full(256, n_bytes / 256.0))
    results["Chi-square"] = ("PASS" if p >= 0.01 else "FAIL", p)

    # 3. Autocorrelation at lag 1
    lag1 = np.corrcoef(bits[:-1], bits[1:])[0, 1]
    results["Autocorr"]   = ("PASS" if abs(lag1) < 0.01 else "FAIL", lag1)

    for test, (result, val) in results.items():
        print(f"  {test:<14} {result}   value={val:.6f}")

drng_compliance_check()
```

```
  Monobit        PASS   value=0.412871
  Chi-square     PASS   value=0.583241
  Autocorr       PASS   value=0.000312
```

Complete Standards Reference for the Intel DRNG

Standard	Title (abbreviated)	DRNG Relevance
NIST SP 800-90A Rev.1	Approved DRBG Mechanisms	CTR_DRBG with AES-256
NIST SP 800-90B	Entropy Source Requirements	Physical entropy source + condition
NIST SP 800-90C (draft)	DRBG Construction	Full pipeline guidance
NIST SP 800-22 Rev.1a	Statistical Test Suite	RDRAND output validation
NIST SP 800-131A Rev.2	Algorithm Transitions	AES-256 approved through 2030+
FIPS 140-3	Cryptographic Module Validation	Module-level certification
FIPS 197	AES Specification	Conditioner and DRBG primitive
ISO/IEC 19790:2012	Module Security Requirements	Basis for FIPS 140-3

Summary: The Intel DRNG Pipeline

Stage	Mechanism	Standard	Output	Instruction
Entropy Source	Metastable latch	SP 800-90B	$\sim$3 Gbps raw	—
Health Tests	RCT + APT	SP 800-90B	Pass/fail flag	—
Conditioner	AES-256-CBC-MAC	SP 800-90B	256-bit clean	—
DRBG	AES-256-CTR	SP 800-90A	128-bit blocks	`RDSEED`
Output	64-bit samples	FIPS 140-3	Random values	`RDRAND`

Key takeaways:

- The DRNG is the **only broadly deployed TRNG in a general-purpose CPU**
- It conforms to SP 800-90A, SP 800-90B, SP 800-22, and FIPS 140-3
- **Use it as an additional entropy source** mixed with OS entropy
- **Never** use RDRAND as the *sole* entropy source in a security-critical application
- **Always check the carry flag** and retry on CF=0
- The CrossTalk/SRBDS vulnerability (2020) underscores the importance of keeping microcode up to date

Further Reading

Primary sources:

- Cox, Dike, Johnston (2011): "Intel's Digital Random Number Generator (DRNG)," *IEEE Hot Chips 23*
- Taylor, Cox (2011): "Behind Intel's New Random-Number Generator," *IEEE Spectrum*
- Hamburg, Kocher, Marson (2012): "Analysis of Intel's Ivy Bridge Digital Random Number Generator," Cryptography Research Inc.
- Intel (2012): "DRNG Software Implementation Guide," Rev. 1.1

Standards:

- NIST SP 800-90A Rev. 1 (2015): `csrc.nist.gov`
- NIST SP 800-90B (2018): `csrc.nist.gov`
- FIPS 140-3 (2019): `csrc.nist.gov/publications/fips/fips140-3`

Security analysis:

- Ragab et al. (2020): "CrossTalk: Speculative Data Leaks Across Cores Are Real," *IEEE S&P 2021*
- Bernstein, Lange: "Failures in NIST's ECC Standards," *IACR ePrint 2014/932*

Questions?