Cryptographic Random Numbers & Secure Hashing

Standards, Algorithms, and Applications

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 Randomness Matters in Cryptography

Core Principle

The security of nearly every cryptographic system depends on the unpredictability of some secret value.

Where randomness is required:

- Symmetric key generation (AES, ChaCha20)
- Asymmetric key generation (RSA primes, ECC scalars)
- Nonces, IVs, and salts
- Challenge–response protocols
- Blinding factors (side-channel protection)

Historical Failure: Debian OpenSSL (2008)

A two-line code patch removed the main entropy source. Effective key space collapsed to $\approx 2^{15}$. All keys generated 2006–2008 were broken.

More Real-World Failures

PS3 Elliptic Curve (2010)

- Sony reused the same nonce k for every ECDSA signature
- Private key recovered algebraically
- Entire console security bypassed

Android Bitcoin Wallets (2013)

- Java `SecureRandom` seeded poorly on Android
- ECDSA nonces repeated
- Private keys extracted; funds stolen

Lesson

Weak randomness is not a theoretical concern. It has resulted in real, exploitable vulnerabilities.

Taxonomy of Random Number Generators

TRNG
True RNG

- Physical process
- Non-deterministic
- Thermal noise
- Radioactive decay
- Photon timing
- Slow; limited rate

PRNG
Pseudo-RNG

- Deterministic
- Seeded state
- Fast output
- Long period
- **NOT crypto-safe** unless specifically designed

CSPRNG
Crypto-Secure PRNG

- Deterministic
- Seeded by TRNG
- **Computationally indistinguishable** from random
- Forward secure
- Backward secure

Requirements for a CSPRNG

A PRNG is **cryptographically secure** if it satisfies:

1. **Next-bit unpredictability:** Given the first k bits, no polynomial-time algorithm can predict bit $k+1$ with probability significantly greater than $\frac{1}{2}$.
2. **State compromise extension resistance:**
   - *Forward security:* Compromising current state reveals nothing about past outputs.
   - *Backward security:* After reseeding, outputs are independent of previous state.
3. **Statistical indistinguishability:** Passes NIST SP 800-22 and Dieharder test suites.
4. **Sufficient entropy at seeding:** Garbage in, garbage out.

Formal Statement

A CSPRNG passes every polynomial-time statistical test if and only if it is computationally indistinguishable from a true random oracle.

Entropy and Information Theory

Shannon Entropy of a discrete source X:

$$H(X) = -\sum_i p_i \log_2 p_i \quad \text{(bits)}$$

- Maximum entropy: uniform distribution, $H = \log_2 n$
- Minimum entropy (min-entropy): $H_\infty(X) = -\log_2 \max_i p_i$
- NIST uses *min-entropy* as the conservative measure

Entropy Sources in Practice:

- Hardware interrupts, disk timing, network packet arrival
- Mouse movement, keystroke timing
- CPU jitter (RDRAND, RDSEED on Intel)
- Dedicated TPM / Hardware Security Module (HSM)

Example 1: Linear Congruential Generator (LCG)

Recurrence:

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

Common parameters (glibc): $a = 1103515245,\ c = 12345,\ m = 2^{31}$

Strengths

- Extremely fast
- Minimal memory
- Long-studied

Cryptographic Weaknesses

- Output is *linear* in internal state
- 3 consecutive outputs suffice to recover all parameters
- State fully predictable after recovery

Verdict

Use for simulations, never for cryptography.

Example 2: Blum–Blum–Shub (BBS)

Setup: Choose primes $p, q \equiv 3 \pmod 4$; set $M = pq$.
Recurrence:

$$X_{n+1} = X_n^2 \bmod M$$

Output the *least significant bit* (or log log M bits) of each X_n.

Security basis:

- Hardness of Quadratic Residuosity Problem (QRP)
- Provably as hard as factoring M
- **Theoretically the strongest security proof of any PRNG**

Practical limitations:

- Modular squaring is expensive: ≈ 1 bit per operation
- $1000\times$ slower than stream cipher CSPRNGs
- Used in proofs and standards references, rarely in production

Example 3: Mersenne Twister (MT19937)

Properties:

- Period: $2^{19937} - 1$
- 623-dimensional equidistribution
- Passes almost all statistical tests
- Default in Python `random`, C++ `mt19937`

NOT a CSPRNG

- Internal state: 624 32-bit words
- Observing 624 outputs recovers full state
- All future *and past* outputs predictable

Rule of Thumb

MT is excellent for Monte Carlo simulations, games, and procedural generation. **Never** use it for keys, nonces, tokens, or passwords.

Example 4: Fortuna (Ferguson & Schneier, 2003)

Design goals: Robust CSPRNG resilient to seed compromise.

Architecture:

- **32 entropy pools** ($P_0 \ldots P_{31}$) accumulate from multiple sources
- Reseed occurs when P_0 holds ≥ 64 bytes; pool P_i used every 2^i reseeds
- Generator: AES-256 in counter mode, rekeyed after each block
- Key is replaced after every 1 MB of output (forward security)

Security properties:

- Slow pools prevent an adversary from controlling all entropy
- Rekeying ensures state compromise does not expose prior output
- Basis of `/dev/urandom` on macOS and FreeBSD

Example 5: ChaCha20-based CSPRNG (Linux)

ChaCha20 stream cipher as PRNG:

- 256-bit key + 64-bit counter + 64-bit nonce
- ARX design (Add-Rotate-XOR): no S-boxes, no lookup tables
- 20 rounds of quarter-round operations

Linux kernel (5.6+) random subsystem:

- Hardware entropy: RDRAND, RDSEED, TPM, disk, network
- Single ChaCha20 CSPRNG; `/dev/urandom` and `getrandom()` are equivalent after boot
- `/dev/random` no longer blocks after init (Linux 5.6)
- `getrandom()` syscall blocks *only* at early boot

Windows Equivalent

`BCryptGenRandom` (CNG) — replaces deprecated `CryptGenRandom`. Uses AES-CTR DRBG internally.

Using CSPRNGs in Python

```python
import secrets   # CSPRNG wrapper -- always use this for crypto
import os

# -- Secure random bytes --
key_256  = secrets.token_bytes(32)     # 256-bit AES key
nonce    = secrets.token_bytes(12)     # 96-bit GCM nonce
salt     = secrets.token_bytes(16)     # 128-bit password salt

# -- Secure random integers --
pin      = secrets.randbelow(10**6)    # 6-digit PIN (uniform)
bits     = secrets.randbits(128)       # 128 random bits

# -- URL-safe token (session IDs, CSRF tokens) --
token    = secrets.token_urlsafe(32)   # 43-char base64url string

# -- Low-level OS interface --
raw      = os.urandom(32)              # same entropy pool

# -- NEVER do this for crypto --
import random
bad_key  = random.getrandbits(256)     # Mersenne Twister -- BROKEN
```

NIST SP 800-90 Series

SP 800-90A Rev.1 — DRBG Mechanisms

- Hash_DRBG (SHA-256, SHA-512)
- HMAC_DRBG (recommended)
- CTR_DRBG (AES-256, recommended)
- Dual_EC_DRBG **(withdrawn – NSA backdoor)**

SP 800-90B — Entropy Sources

- Min-entropy assessment methodology
- Statistical tests for hardware sources
- Health test requirements

SP 800-90C — DRBG Construction

- How to combine entropy source with DRBG

Dual_EC_DRBG

Standardized 2006.
Snowden documents (2013) revealed probable NSA backdoor via rigged elliptic curve points.
Never use.

NIST SP 800-22: Statistical Testing

15 statistical tests for evaluating RNG output:

- Frequency (monobit)
- Frequency within block
- Runs test
- Longest run of ones
- Binary matrix rank
- Spectral (DFT)
- Non-overlapping template
- Overlapping template
- Maurer's universal
- Linear complexity
- Serial test
- Approximate entropy
- Cumulative sums
- Random excursions (2)

Important Caveat

Passing SP 800-22 is *necessary but not sufficient* for a CSPRNG. LCG can pass many of these tests. Computational indistinguishability is a stronger requirement.

Cryptographic Hash Functions: Definition

A hash function $H : \{0,1\}^* \to \{0,1\}^n$ maps arbitrary-length input to a fixed n-bit digest.

Required security properties:

1. **Pre-image resistance (one-way):** Given y, infeasible to find x such that $H(x) = y$. Cost: $\mathcal{O}(2^n)$.
2. **Second pre-image resistance:** Given x, infeasible to find $x' \neq x$ such that $H(x') = H(x)$. Cost: $\mathcal{O}(2^n)$.
3. **Collision resistance:** Infeasible to find any pair (x, x') with $x \neq x'$ and $H(x) = H(x')$. Cost: $\mathcal{O}(2^{n/2})$ by birthday bound.

Avalanche Effect

Flipping one input bit changes $\approx 50\%$ of output bits. This is essential for hiding partial information.

MD5: A Broken Hash

Specification:

- Designed by Ron Rivest, 1992
- Output: 128 bits
- Merkle–Damgård construction

Attack timeline:

- 1996: Dobbertin finds weaknesses
- 2004: Wang et al. — full collision in hours
- 2008: Rogue CA certificate forged using MD5 collision
- 2012: Flame malware exploited MD5 collision for code signing

Status: **BROKEN**

- Collisions found in < 1 second on a laptop
- Do not use for *any* security purpose
- Acceptable only for non-security checksums (e.g., file deduplication where adversary is not present)

SHA-1: Deprecated

Specification:

- Designed by NSA, published 1995
- Output: 160 bits
- Merkle–Damgård construction

Attack timeline:

- 2005: Wang et al. — theoretical collision (2^{69})
- 2017: SHAttered (Google/CWI) — practical collision ($2^{63.1}$ SHA-1 calls, 110 GPU-years)
- 2019: Chosen-prefix collision — $\approx$ 100k USD on cloud GPUs
- 2020: Browsers drop SHA-1 TLS certificates

Status: **DEPRECATED**

- Collision resistance broken
- NIST deprecated for digital signatures (2015)
- Acceptable only in legacy HMAC-SHA1 (pre-image still holds)
- Migrate all new code to SHA-256 or SHA-3

SHA-2 Family

Standardized: FIPS 180-4 (2015)
Construction: Merkle–Damgård with Davies–Meyer compression

Variant	Output	Block	Word	Rounds
SHA-224	224 bits	512 bits	32-bit	64
SHA-256	256 bits	512 bits	32-bit	64
SHA-384	384 bits	1024 bits	64-bit	80
SHA-512	512 bits	1024 bits	64-bit	80
SHA-512/224	224 bits	1024 bits	64-bit	80
SHA-512/256	256 bits	1024 bits	64-bit	80

- **No known practical attacks** on collision resistance
- SHA-256 is the workhorse: TLS, Bitcoin, code signing
- SHA-512/256 faster on 64-bit CPUs; same security as SHA-256
- Length-extension attack possible — use HMAC, not raw SHA-2

SHA-3 / Keccak

Standardized: FIPS 202 (2015)
Construction: Sponge construction — fundamentally different from SHA-2

Sponge phases:

1. **Absorb:** XOR input blocks into state, apply permutation f
2. **Squeeze:** Extract output blocks from state

SHA-3 Variants:

- SHA3-224, SHA3-256, SHA3-384, SHA3-512
- SHAKE128, SHAKE256 (XOFs — variable output)

Advantages:

- **Immune to length-extension attacks**
- Structurally independent of SHA-2
- SHAKE variants allow arbitrary-length output (KDF use)

BLAKE2 and BLAKE3

BLAKE2 (2012) — Not a NIST standard but widely deployed:

- BLAKE2b: 512-bit output, optimized for 64-bit platforms
- BLAKE2s: 256-bit output, optimized for 32-bit / embedded
- **Faster than MD5 in software**; security $\geq$ SHA-3
- No length-extension vulnerability
- Built-in keying (MAC), tree hashing, personalization
- Used in: WireGuard, Argon2, Zcash, libsodium

BLAKE3 (2020):

- Merkle tree structure — massively parallelizable
- Single algorithm: hashing, KDF, PRF, XOF
- Typically 3–10$\times$ faster than BLAKE2 on multi-core
- **Not yet in NIST standards** — evaluate before use in regulated environments

Hashing in Python

```python
import hashlib

msg = b"Graduate CE -- Cryptography Lecture"

# SHA-256 (most common)
h256 = hashlib.sha256(msg).hexdigest()
print(f"SHA-256 : {h256}")

# SHA-3-256 (sponge, no length extension)
h3   = hashlib.sha3_256(msg).hexdigest()
print(f"SHA3-256: {h3}")
```

Hashing in Python (Cont.)

```python
# SHA-512
h512 = hashlib.sha512(msg).hexdigest()
print(f"SHA-512 : {h512}")

# BLAKE2b (fast, keyed)
hb2 = hashlib.blake2b(msg, key=b"secret-key-here").hexdigest()
print(f"BLAKE2b : {hb2}")

# SHAKE-256 with variable output (XOF)
shake = hashlib.shake_256(msg).hexdigest(64)  # 64 bytes = 512 bits
print(f"SHAKE256: {shake}")
```

Message Authentication Codes (MACs)

Problem: A hash alone only provides integrity, not *authentication* — anyone can compute $H(m)$.

MAC: $\text{MAC}(k, m)$ — keyed primitive that provides:

- **Integrity:** Any modification to m is detected
- **Authenticity:** Only parties with key k can generate valid tag

Naive construction: $H(k\|m)$ or $H(m\|k)$

- $H(k\|m)$: Vulnerable to length-extension attack on SHA-2
- $H(m\|k)$: Vulnerable to collision attack if H has weak collision resistance

Solution: HMAC

$$\text{HMAC}(k, m) = H\big((k' \oplus \text{opad})\|H((k' \oplus \text{ipad})\|m)\big)$$

Two-pass construction provably secure if H is a PRF.

HMAC in Python

```python
import hmac
import hashlib
import secrets

key = secrets.token_bytes(32)              # 256-bit secret key
msg = b"Authenticated payload: user=alice"

# Compute HMAC-SHA256
tag = hmac.new(key, msg, hashlib.sha256).hexdigest()
print(f"Tag: {tag}")

# Verify -- constant-time comparison prevents timing attacks
def verify_hmac(key, msg, received_tag):
    expected = hmac.new(key, msg, hashlib.sha256).digest()
    received = bytes.fromhex(received_tag)
    return hmac.compare_digest(expected, received)   # timing-safe

ok = verify_hmac(key, msg, tag)
print(f"Valid: {ok}")

# NEVER do this -- timing oracle
# if hmac_tag == computed_tag:  <-- vulnerable to timing attack
```

Hash Function Standards Reference

Algorithm	Standard	Output	Status
MD5	RFC 1321	128 bit	**Broken**
SHA-1	FIPS 180-4	160 bit	**Deprecated**
SHA-256	FIPS 180-4	256 bit	**Current**
SHA-384	FIPS 180-4	384 bit	**Current**
SHA-512	FIPS 180-4	512 bit	**Current**
SHA3-256	FIPS 202	256 bit	**Current**
SHA3-512	FIPS 202	512 bit	**Current**
SHAKE128	FIPS 202	variable	**Current**
SHAKE256	FIPS 202	variable	**Current**
BLAKE2b	RFC 7693	512 bit	**Recommended**
BLAKE3	(no NIST std.)	variable	**Evaluate**
HMAC-SHA256	FIPS 198-1	256 bit	**Current**

CSPRNG Standards Reference

Mechanism	Standard	Status
Hash_DRBG (SHA-256)	NIST SP 800-90A	**Approved**
HMAC_DRBG	NIST SP 800-90A	**Approved**
CTR_DRBG (AES-256)	NIST SP 800-90A	**Approved**
Dual_EC_DRBG	SP 800-90A (removed)	**Withdrawn**
Fortuna	Academic / macOS	**Widely used**
ChaCha20 CSPRNG	Linux kernel / libsodium	**Widely used**
BCryptGenRandom	Windows CNG	**Approved**
/dev/urandom	POSIX (Linux/macOS)	**Standard**
LCG / MT19937	—	**Not for crypto**

FIPS 140-3 validation requires use of SP 800-90A approved DRBGs.

Applications of Secure Hashing and CSPRNGs

Hashing Applications

- Digital signatures (sign $H(m)$, not m)
- Password storage: Argon2, bcrypt, scrypt (iterated hash)
- Certificate fingerprints (TLS)
- Blockchain / Merkle trees
- File integrity and deduplication
- Commitment schemes

CSPRNG Applications

- Symmetric key generation
- RSA/ECC key pair generation
- Session tokens and CSRF tokens
- Nonce/IV generation for AEAD
- Salt generation for password hashing
- OTP / TOTP seed generation

Common Implementation Mistakes

1. **Using `random` module for security-sensitive values**
   Always use `secrets` or `os.urandom`.
2. **Comparing MAC tags with ==**
   Use `hmac.compare_digest` to prevent timing oracle attacks.
3. **Hashing passwords with a plain hash (MD5/SHA-256)**
   Use a memory-hard KDF: Argon2id, bcrypt, or scrypt.
4. **Reusing nonces / IVs with the same key**
   Use a CSPRNG per message, or a deterministic nonce counter.
5. **Length-extension on raw SHA-2 for MAC construction**
   Use HMAC or SHA-3, never $H(k \| m)$ with SHA-2.
6. **Seeding a PRNG with time (e.g., `srand(time(NULL))`)**
   Timestamp entropy is very low; easily guessed.
7. **Truncating hashes arbitrarily**
   Security drops proportionally. Use a standard variant instead.

Algorithm Selection Guide

Choose the right primitive:

Need	Use
Random bytes (crypto)	`secrets.token_bytes(n)`
Random bytes (FIPS-validated)	CTR_DRBG (AES-256) per SP 800-90A
General purpose hash	SHA-256 or SHA3-256
Hash on 64-bit platform (performance)	BLAKE2b or SHA-512/256
Variable-length output (KDF/XOF)	SHAKE-256 or BLAKE3
Message authentication (MAC)	HMAC-SHA256 or BLAKE2b (keyed)
Password hashing	Argon2id $\rightarrow$ bcrypt $\rightarrow$ scrypt
Digital signature hashing	SHA-256 (RSA/ECDSA) or SHA3-256
FIPS 140-3 regulated environment	SHA-256/SHA-512, HMAC-SHA256, CTR_DRBG

Key Takeaways

Cryptographic Randomness

- Only CSPRNGs (e.g., ChaCha20, CTR_DRBG) are safe for crypto
- LCG and Mersenne Twister are **not** cryptographically secure
- Seed quality matters — use hardware entropy sources
- NIST SP 800-90A defines approved DRBG constructions

Secure Hashing

- MD5 is **broken**; SHA-1 is **deprecated**
- SHA-256 and SHA3-256 are the current workhorses
- Use HMAC or SHA-3 to avoid length-extension attacks
- BLAKE2b is fast, secure, and widely deployed

Engineering Practice

- Use high-level APIs (`secrets`, `hmac`, `hashlib`)

References and Further Reading

- **NIST SP 800-90A Rev.1** — Recommendation for Random Number Generation Using Deterministic RBGs (2015)
- **NIST SP 800-90B** — Recommendation for Entropy Sources Used for RBGs (2018)
- **FIPS 180-4** — Secure Hash Standard (SHA-1, SHA-2) (2015)
- **FIPS 202** — SHA-3 Standard: Permutation-Based Hash and XOF (2015)
- **FIPS 198-1** — The Keyed-Hash Message Authentication Code (HMAC) (2008)
- **RFC 7693** — The BLAKE2 Cryptographic Hash and MAC (2015)
- N. Ferguson, B. Schneier, T. Kohno — *Cryptography Engineering*, Wiley, 2010
- D. Boneh, V. Shoup — *A Graduate Course in Applied Cryptography* (freely available online)
- *SHAttered* — `shattered.io` (2017 SHA-1 collision)

Questions?