Published Date: - 25-11-2025
Page No: - 176-181

RANDOM NUMBER GENERATION IN COMPUTING: SECURITY, ARCHITECTURE AND OS-LEVEL IMPLEMENTATIONS

Nuriddin Safoev
Tashkent University of Information Technologies named after Muhammad al-Khwarizmi, Tashkent, Uzbekistan

ABSTRACT

Random number generation (RNG) is a cornerstone of secure computing, underpinning cryptography, system security, and reliable software operations. Modern operating systems incorporate intricate mechanisms to produce high-quality randomness suitable for a variety of applications. This paper surveys the RNG architectures implemented in leading operating systems, including Microsoft Windows, Linux, and Apple's macOS/iOS. It examines entropy sources, cryptographically secure deterministic random bit generators (DRBGs), system APIs, and methods for evaluating randomness quality. The analysis emphasizes architectural differences, identifies potential vulnerabilities, and outlines best practices for secure randomness generation. The paper serves as a reference for students, software developers, and security professionals seeking an in-depth comparative understanding of operating system-level RNG strategies.

KEYWORDS: Random Number Generation, Operating Systems, Entropy, DRBG, Cryptography, RNG APIs, Windows, Linux, macOS.

INTRODUCTION

Randomness is a fundamental component across diverse computing domains, including simulations, gaming, probabilistic algorithms, and, critically, cryptographic and security systems. In cryptography, the unpredictability of generated values directly determines the strength of keys, initialization vectors (IVs), nonces, salts, and secure tokens. High-quality randomness ensures that sensitive data remains confidential, resistant to manipulation, and secure against adversarial analysis [1].

The phrase "security relies on randomness" captures this dependency, as predictable or weak random values can compromise entire systems [2]. Essential use cases include:

- Encryption keys: Must be unpredictable to prevent unauthorized access.
- Nonces: One-time numbers preventing replay attacks and ensuring session uniqueness.
- Salts: Random values appended to passwords, mitigating precomputed hash attacks.
- Secure tokens: Critical for authentication, session tracking, and secure communications.

Insufficient entropy in random numbers has historically resulted in severe security breaches. Well-known incidents, such as flawed key generation, illustrate that weak RNGs can lead to compromised encryption, token forgery, and impersonation [3].

High-quality randomness is crucial for security protocols like:

- TLS (Transport Layer Security): Secures web communications.
- VPNs (Virtual Private Networks): Protects network traffic.
- Digital signatures: Authenticate and preserve data integrity.

During operations like TLS handshakes, random values are used to generate session keys. Predictable values in such contexts could allow attackers to decrypt sensitive communications. Operating systems address these needs by integrating RNG subsystems that gather entropy from multiple sources, including:

- User interactions (keyboard, mouse).
- Hardware timing variations (network packets, disk access).
- Hardware-based noise sources (Intel RDRAND, Trusted Platform Modules).

Collected entropy is processed through cryptographic algorithms—such as AES-CTR, SHA-based DRBGs, or ChaCha20—to produce secure pseudorandom streams. Standardized APIs (e.g., Windows' BCryptGenRandom, Linux's /dev/random, /dev/urandom, getrandom(), and macOS's arc4random() or SecRandomCopyBytes()) enable developers to safely access random values without implementing custom RNGs [4].

While the goal—high-quality randomness—is consistent across platforms, design approaches vary in:

- Platform architectures: Windows employs CNG, Linux relies on device files and system calls, and macOS/iOS integrates RNGs into libraries and hardware.
- Entropy sourcing: Some systems rely more on hardware RNGs; others emphasize software and event-driven inputs.
- DRBG selection: Implementations may use AES, SHA, HMAC, or ChaCha20 depending on performance and security requirements.

This paper provides a comparative overview of RNG implementation strategies in major operating systems, highlighting best practices, potential weaknesses, and avenues for future improvements.

2. Background and Importance

Random number generation underpins modern cryptography, where the unpredictability and entropy of values are directly tied to data confidentiality and system integrity. RNGs are used to produce keys, initialization vectors, nonces, salts, session IDs, and digital signatures. Predictable outputs jeopardize the entire security framework [5].

Real-world incidents, such as the Debian OpenSSL vulnerability (2006–2008), demonstrate the catastrophic effects of weak RNGs. A modification that removed key entropy collection reduced possible key variations to 32,768, making brute-force attacks trivial. This affected SSH keys, SSL certificates, and other secure components, highlighting the importance of careful RNG design.

To address these challenges, modern operating systems use layered RNG architectures:

- Entropy sources: Non-deterministic hardware and software inputs, such as keystroke timing, disk I/O, or hardware RNGs (e.g., Intel RDRAND, AMD RdSeed, TPMs).
- Entropy pools: Aggregates of raw entropy buffered until sufficient for secure seeding of DRBGs. Linux distinguishes pools for /dev/random and /dev/urandom to prevent insufficient randomness.
- DRBGs/PRGs: Expand limited entropy into secure pseudorandom streams using AES-CTR, HMAC, SHA, or ChaCha20. Compliance with NIST SP 800-90A/B/C ensures cryptographic robustness.
- APIs: Expose secure randomness to developers. Correct usage is critical, as weak alternatives (e.g., rand()/srand()) can compromise security.

Well-engineered RNGs protect against weak encryption, token forgery, impersonation, and unauthorized access. Continuous improvements and audits are necessary to maintain security in evolving threat landscapes.

3. Random Number Generation in Operating Systems

3.1. RNG in Microsoft Windows

Windows RNG is implemented via the Cryptography API: Next Generation (CNG), incorporating an AES-CTR-based DRBG compliant with NIST SP 800-90A.

Entropy sources include:

- Hardware RNGs (e.g., Intel RDRAND).
- System events (interrupts, calls).
- User input (mouse, keyboard).
- Network traffic timing.
- Disk read/write delays.

APIs and components:

- SystemPrng: Kernel-level CSPRNG, periodically reseeded.
- BCryptGenRandom(): Provides developers access to secure random bytes, with options to use system-preferred or custom RNG algorithms.

Security features:

- Forward and backward secrecy through periodic reseeding.
- Entropy monitoring to detect depletion or tampering.

3.2. RNG in Linux Systems

Linux provides /dev/random (blocking) and /dev/urandom (non-blocking) interfaces, supplemented by the getrandom() syscall. Since kernel 5.6, ChaCha20-based DRBGs replace older SHA-1-based generators.

Entropy collection:

- Interrupt timings, keyboard/mouse events.
- Device driver noise (disk, network).
- Hardware RNGs (mixed to prevent bias).

The rngd daemon integrates hardware RNG data, ensuring FIPS-compliant validation and proper entropy injection.

3.3. RNG in Apple's macOS and iOS

Apple's RNG combines hardware acceleration via the Secure Enclave and T2 Security Chip with software-based entropy.

APIs and DRBGs:

- SecRandomCopyBytes(): Primary interface for secure random bytes.
- AES-CTR and ChaCha20-based DRBGs ensure cryptographic security.
- Frequent reseeding and sandboxing reinforce forward/backward secrecy.

Applications: Secure Boot, FileVault encryption, biometric authentication, and other critical security functions.

4. Comparative Overview

- Microsoft Windows: Utilizes the Cryptography API: Next Generation (CNG) with an AES-CTR Deterministic Random Bit Generator (DRBG). It draws entropy from diverse sources, including hardware RNGs (e.g., Intel's RDRAND), system event timings, user inputs, network activity, and disk operations. The BCryptGenRandom() API provides developers with access to

cryptographically secure random bytes, with periodic reseeding ensuring forward and backward secrecy.

- Linux: Employs a dual-interface model with /dev/random (blocking) and /dev/urandom (non-blocking), supplemented by the modern getrandom() system call. Since kernel 5.6, it uses a ChaCha20-based DRBG, collecting entropy from interrupt timings, user inputs, device drivers, and hardware RNGs. The rngd daemon enhances hardware entropy integration, ensuring quality through validation.
- Apple macOS/iOS: Integrates RNG with hardware acceleration via the Secure Enclave and T2 Security Chip, using the SecRandomCopyBytes() API. It combines AES-CTR and ChaCha20-based DRBGs, sourcing entropy from system events and dedicated hardware. Sandboxing and frequent reseeding enhance security, supporting critical functions like Secure Boot, FileVault, and biometric authentication.

Table 1: Comparison of RNGs for operation systems

Feature	Windows	Linux	macOS/iOS
DRBG Type	AES-CTR (NIST SP 800-90A)	ChaCha20	AES-CTR / ChaCha20
Entropy Sources	Hardware RNG, system events, user input, network, disk	Interrupts, user input, drivers, hardware RNG	Secure Enclave, T2 Chip, system events
Developer API	BCryptGenRandom()	/dev/random, /dev/urandom, getrandom()	SecRandomCopyBytes()
Reseeding	Periodic for forward/backward secrecy	Automatic, kernel-managed	Frequent, hardware-assisted
Security Emphasis	Enterprise and general-purpose computing	Kernel and user-space cryptography	Mobile, FIPS-compliant, hardware-accelerated

Shared principles:

- Secure entropy acquisition from diverse sources.
- Cryptographically compliant DRBG expansion.
- Regular reseeding to maintain secrecy.
- Accessible APIs for developers.

Challenges:

- Low entropy at system startup.
- Embedded systems with limited hardware.
- Heterogeneous hardware environments.

Future directions:

- Quantum-resistant RNG designs.
- Enhanced entropy validation and monitoring.
- Open-source transparency and third-party audits.

Conclusions

The security and reliability of modern computing systems depend on robust random number generation. Windows, Linux, and macOS/iOS each implement unique RNG architectures tailored to their environments while sharing core principles of secure entropy collection, cryptographic expansion, and accessible APIs. Continued innovation in RNG design is vital to

counter increasingly sophisticated attacks and ensure that cryptographic mechanisms remain resilient and trustworthy.

References

1. Barker, E., & Kelsey, J. (2015). Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised). NIST Special Publication 800-90A Rev. 1. https://doi.org/10.6028/NIST.SP.800-90Ar1
2. Eastlake, D., Schiller, J., & Crocker, S. (2005). Randomness Requirements for Security. RFC 4086. https://www.rfc-editor.org/rfc/rfc4086
3. Microsoft. (2023). Cryptography API: Next Generation. Microsoft Docs. https://learn.microsoft.com/en-us/windows/win32/seccng/cng-portal
4. Microsoft. (2023). BCryptGenRandom function (bcrypt.h). Microsoft Docs. https://learn.microsoft.com/en-us/windows/win32/api/bcrypt/nf-bcrypt-bcryptgenrandom
5. Linux Kernel Documentation. (2023). Random Number Generator. https://www.kernel.org/doc/html/latest/admin-guide/dev-random.html
6. Linux man-pages project. (2023). getrandom(2) – Linux manual page. https://man7.org/linux/man-pages/man2/getrandom.2.html
7. Apple Developer Documentation. (2023). SecRandomCopyBytes. https://developer.apple.com/documentation/security/1399291-secrandomcopybytes
8. Apple. (2020). Platform Security Guide. https://support.apple.com/guide/security/welcome/web
9. Gutterman, Z., Pinkas, B., & Reinman, T. (2006). Analysis of the Linux Random Number Generator. IEEE Symposium on Security and Privacy. https://doi.org/10.1109/SP.2006.26
10. Dorrendorf, L., Gutterman, Z., & Pinkas, B. (2007). Cryptanalysis of the Random Number Generator of the Windows Operating System. ACM CCS. https://doi.org/10.1145/1315245.1315274
11. Lacharme, P. (2012). Security flaws in Linux's /dev/random. https://eprint.iacr.org/2012/251
12. BSD Unix. (2022). arc4random and related APIs. https://man.openbsd.org/arc4random
13. Kelsey, J., Schneier, B., Ferguson, N. (1999). Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator. https://www.schneier.com/paper-yarrow.pdf
14. Dodis, Y., et al. (2013). Security Analysis of Pseudorandom Number Generators with Input: /dev/random is not Robust. ACM CCS. https://doi.org/10.1145/2508859.2516661
15. Intel Corporation. (2014). Intel® Digital Random Number Generator (DRNG) Software Implementation Guide. https://www.intel.com/content/www/us/en/content-details/671488/intel-digital-random-number-generator-drng-software-implementation-guide.html
16. National Institute of Standards and Technology. (2012). A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. NIST SP 800-22 Rev. 1a. https://doi.org/10.6028/NIST.SP.800-22r1a

17. Müller, T. (2013). Security of the OpenSSL PRNG. International Journal of Information Security, 12(4), 251–265. https://doi.org/10.1007/s10207-013-0213-7

18. Debian Security Advisory. (2008). Debian OpenSSL Predictable PRNG Vulnerability (DSA-1571). https://www.debian.org/security/2008/dsa-1571