SECURE RANDOMNESS IN MODERN OPERATING SYSTEMS: AN ANALYSIS OF RNG ARCHITECTURES IN WINDOWS, LINUX, AND MACOS/IOS

Nuriddin Safoev
Tashkent University of Information Technologies named after Muhammad al-Khwarizmi, Tashkent, Uzbekistan
Fayziraxmonov Boburjon Baxtiyorjon o'g'li
Tashkent University of Information Technologies named after Muhammad al-Khwarizmi, Tashkent, Uzbekistan
https://doi.org/10.5281/zenodo.15637151

Abstract. Random number generation (RNG) plays a foundational role in security, cryptography, and system design. Operating systems today implement complex mechanisms for generating random numbers securely. This survey paper presents an overview of RNG techniques used in major operating systems, including Microsoft Windows, Linux, and macOS. We examine entropy sources, deterministic random bit generators (DRBGs), system APIs, and quality testing mechanisms. The survey highlights key differences between OS-level RNG designs and emphasizes best practices, challenges, and potential vulnerabilities. This work aims to serve as a reference for students, developers, and security professionals seeking a comparative understanding of secure randomness in computing environments.

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

1. Introduction

Random number generation plays a pivotal role across numerous domains in computing, such as simulations, gaming applications, probabilistic algorithms, and—most importantly—cryptographic and security systems. In these areas, randomness goes beyond simple variation; it forms the very foundation of unpredictability, which is essential to the strength and reliability of cryptographic mechanisms. The effectiveness of secure cryptographic infrastructures heavily depends on the availability of high-entropy random numbers to maintain data confidentiality, safeguard integrity, and prevent adversaries from deducing or replicating sensitive values [1].

The phrase "security depends on randomness" aptly captures the essence of modern cryptography, where resilience to analysis and guessing is a fundamental requirement [2]. Key components include:

- **Encryption Keys**: These are derived from random data to ensure uniqueness and unpredictability. If keys can be anticipated, the secrecy of encrypted messages is jeopardized.
- **Nonces (Numbers Used Once)**: These ensure that each encryption operation is distinct. If reused or generated predictably, nonces can open the door to replay attacks or breach the consistency of communication.
- **Salts**: Random values appended to passwords prior to hashing, salts defend against precomputed hash attacks, such as those utilizing rainbow tables. Their absence allows attackers to leverage known hash dictionaries to crack passwords.
- **Secure Tokens**: Employed in scenarios like session tracking or password recovery, tokens must be generated with strong entropy to avoid unauthorized access or token forgery.

When randomness is insufficient or flawed, security collapses—often with severe real-world consequences. Attackers may successfully guess keys, recreate secure tokens, or impersonate users within encrypted sessions [3]. Thus, incorporating robust and unpredictable random number generators (RNGs) into secure system design is not optional—it's imperative.

Various widely used security protocols rely on high-quality random numbers, including:

- **TLS (Transport Layer Security)**, which encrypts web communications (e.g., HTTPS),
- **VPNs (Virtual Private Networks)**, which protect internet traffic,
- **Digital Signature mechanisms**, which authenticate and preserve data integrity.

These protocols utilize randomness during crucial steps such as key exchanges, session initialization, and digital signing. For example, during the TLS handshake, random values exchanged between client and server are essential for securely deriving session keys. If an attacker could predict these values, encrypted communications could be compromised.

To support these cryptographic needs, modern operating systems—such as Windows, Linux, and macOS—embed RNG subsystems into their core architecture. These RNGs accumulate entropy from a wide range of unpredictable system-level sources, including:

- User interactions (e.g., mouse movement, keyboard activity),
- Timing discrepancies in hardware behavior (e.g., hard drive access, network packet timing),
- Hardware-based noise sources (e.g., Intel's RDRAND, Trusted Platform Modules).

This raw entropy is then processed by secure algorithms—such as AES in Counter mode, SHA family constructions, or ChaCha20—to expand it into long streams of cryptographically strong pseudorandom bits.

To facilitate widespread and error-free usage, operating systems expose standardized APIs that allow applications and developers to retrieve random values without needing to implement their own potentially insecure methods [4]. Examples include Windows' BCryptGenRandom from the Cryptography API: Next Generation (CNG), Linux's /dev/random, /dev/urandom, and getrandom() system call, as well as macOS's arc4random() and SecRandomCopyBytes() functions.

Although the ultimate purpose—providing secure, high-quality randomness—is shared across platforms, their internal architectures and cryptographic approaches differ based on:

- **Platform-Specific Architectures**: Windows employs the CNG framework, Linux uses character devices and syscalls, and macOS embeds RNG functionality into libraries like libSystem.
- **Entropy Sourcing Approaches**: Some systems lean more heavily on hardware RNGs, while others emphasize software-based and event-driven sources.
- **DRBG Compliance**: Many systems implement NIST SP 800-90A-compliant Deterministic Random Bit Generators (DRBGs), utilizing AES, SHA-256, or HMAC-based designs. Others, especially those optimized for speed or mobile environments, prefer stream ciphers like ChaCha20.

This paper presents a comparative analysis of RNG implementation strategies in major operating systems. It explores the architectural frameworks, entropy acquisition techniques, DRBG models, and accessible APIs that together form the backbone of secure randomness. By evaluating these systems, we aim to highlight best practices, pinpoint vulnerabilities, and propose directions for future enhancements in cryptographic randomness generation.

2. Background and Importance

Random number generation (RNG) serves as a cornerstone in today's computing landscape, particularly within **cryptographic systems**, where the effectiveness and resilience of security mechanisms are heavily reliant on the quality of randomness. Whether it's for generating **encryption keys**, **initialization vectors**, **nonces**, **salts**, **session identifiers**, or **digital signatures**, the confidentiality and integrity of data hinge on producing values that are both **unpredictable and rich in entropy**. If the RNG process is compromised or produces predictable outputs, the entire cryptographic framework becomes vulnerable to exploitation [5].

The risks posed by flawed RNGs have been highlighted by real-world incidents, one of the most infamous being the **Debian OpenSSL vulnerability** between 2006 and 2008. Due to a modification made by a Debian developer—who removed key entropy-collecting functionality—OpenSSL's randomness was significantly weakened. This oversight limited the possible number of cryptographic keys to just **32,768**, rendering them trivially guessable via brute-force attacks. The aftermath affected **SSH keys, SSL certificates**, and other secure components across countless systems, leading to widespread key regeneration and infrastructure re-securing efforts. This incident underscores the **severe consequences of mismanaged entropy and flawed RNG design**.

To mitigate such vulnerabilities, contemporary operating systems and cryptographic libraries adopt **multi-layered RNG architectures** comprising the following core components [6]:

- **Entropy Sources**: These are the foundational inputs for randomness, derived from non-deterministic behaviors in hardware and software. Examples include variations in **keystroke timing**, **mouse movements**, **disk access delays**, and specialized **hardware noise generators** like Intel's **RDRAND**, AMD's **RdSeed**, or dedicated **TPMs (Trusted Platform Modules)** and **HWRNGs (Hardware RNGs)**. The quality and diversity of these inputs are crucial, as they determine the system's overall randomness strength.
- **Entropy Pools**: Operating systems aggregate raw entropy from various sources into internal pools. These act as buffers, collecting and managing entropy until there's enough to securely seed a deterministic generator. For instance, **Linux** maintains distinct pools for **/dev/random** and **/dev/urandom**, using **entropy accounting** to decide when to block or return data. This pooling strategy ensures the system doesn't prematurely emit insufficiently random output.
- **Pseudorandom Number Generators (PRGs)** or **Deterministic Random Bit Generators (DRBGs)**: These cryptographic mechanisms expand limited, high-quality entropy into large quantities of pseudorandom bits. Widely used DRBGs are based on constructions like **AES in Counter mode (AES-CTR)**, **HMAC-based methods**, and **secure hash algorithms** (e.g., SHA-based). More recently, **ChaCha20**, a stream cipher, has been adopted for its performance and security benefits. Most implementations adhere to **NIST SP 800-90A/B/C** guidelines to ensure compliance and cryptographic integrity.
- **Application Programming Interfaces (APIs)**: Operating systems expose secure RNG functionality through well-defined APIs, simplifying developer access while enforcing secure practices. Examples include **BCryptGenRandom** and **RtlGenRandom** on **Windows**,

/dev/random, **/dev/urandom**, and **getrandom()** on **Linux**, and **arc4random()** and **SecRandomCopyBytes** on **macOS**. The proper use of these APIs is critical, as falling back on weak alternatives (e.g., C's rand() or srand()) can introduce severe vulnerabilities into software applications.

The **security and reliability** of cryptographic operations depend fundamentally on the robustness of RNG systems. Failures at this level can result in **broken encryption**, **token forgery**, **impersonation**, or **unauthorized access** to sensitive data. Consequently, random number generators must be **carefully engineered, reviewed, and continually improved** to withstand evolving attack vectors. With this in mind, operating system vendors and security experts consistently enhance RNG subsystems to safeguard the cryptographic processes they support.

3. Random Number Generation in Operating Systems

3.1. RNG in Microsoft Windows

Microsoft Windows incorporates a sophisticated cryptographic system called Cryptography API: Next Generation (CNG) to manage cryptographic functions, including random number generation (RNG). Central to its RNG mechanism is a Deterministic Random Bit Generator (DRBG) that uses the AES block cipher in Counter (CTR) mode, adhering to NIST SP 800-90A standards. This design ensures compliance with federal cryptographic requirements and delivers robust randomness for secure communications and system operations.

Entropy Collection

Windows gathers entropy from diverse hardware and software sources to maintain a robust entropy pool, enhancing unpredictability and resilience against depletion or attacks:

- **Hardware RNGs**: When available, Windows leverages hardware-based random number generators, such as Intel's RDRAND instruction, for high-speed, high-quality entropy.
- **System Event Timing**: Variability in low-level system events, like interrupts and system calls, is captured to add randomness based on system and user activity.
- **User Input**: Timing data from mouse movements and keyboard inputs provides entropy, particularly during system boot or when entropy is limited.
- **Network Activity**: Randomness is derived from variations in network packet arrival times and traffic patterns.
- **Disk Operations**: Read/write latencies from storage devices contribute additional entropy.

This multi-faceted approach ensures sufficient entropy collection across various environments, including servers without user interaction.

Core RNG Components and APIs

Windows provides several components and interfaces for random number generation, serving both system-level needs and application developers:

- **SystemPrng**: The kernel's primary pseudorandom number generator, it supplies cryptographically secure random data to system components and periodically reseeds to maintain security features like forward and backward secrecy.
- **BCryptGenRandom()**: A key API for developers, enabling both user-mode and kernel-mode applications to access secure random data. It offers two options:

- Using the system's default RNG (via the BCRYPT_USE_SYSTEM_PREFERRED_RNG flag).
- Specifying a custom algorithm, such as a hardware RNG. This API is widely used for generating encryption keys, nonces, session tokens, and salts within the Windows ecosystem.

Security Features and Reseeding

Windows CNG prioritizes strong security through:

- **Forward Secrecy**: Ensures that a compromised RNG state at a given time cannot expose prior outputs, achieved through regular reseeding with fresh entropy.
- **Backward Secrecy**: Protects future outputs after a state compromise by continuously injecting new entropy.
- **Entropy Monitoring**: The DRBG actively monitors entropy quality, triggering reseeding or failure if sources weaken, safeguarding against entropy starvation or manipulation.

In conclusion, Windows' RNG system, integrated within its CNG framework, combines hardware entropy, system noise, and strict NIST compliance to deliver highly secure randomness for diverse computing needs.

3.2. RNG in Linux Systems

Linux employs a robust and adaptable random number generation (RNG) architecture critical for system security and cryptographic operations. Designed to balance performance, reliability, and cryptographic strength, it collects entropy from diverse sources to provide secure pseudorandom data for both kernel and user-space applications.

Architecture Overview

The Linux kernel uses a dual-RNG interface centered on two device files: /dev/random and /dev/urandom, each serving distinct purposes:

- **/dev/random**: A blocking interface that only outputs data when sufficient entropy is available, ideal for high-security tasks like cryptographic key generation.
- **/dev/urandom**: A non-blocking interface that delivers random bytes using a cryptographically secure pseudorandom number generator (CSPRNG), even with limited entropy. Modern improvements have minimized security differences between the two.

Since Linux kernel 5.6, the RNG has adopted a ChaCha20-based Deterministic Random Bit Generator (DRBG), replacing the older SHA-1-based algorithm. ChaCha20 provides superior performance and resistance to cryptanalytic attacks, making it suitable for both kernel and user applications.

Entropy Collection

Linux gathers entropy from various system activities to maintain a robust entropy pool:

- **Interrupt Timings**: Asynchronous interrupt timings provide unpredictable data for entropy.
- **User Input**: Timing of keyboard and mouse events, particularly during boot, contributes to entropy.
- **Device Drivers**: Timing and noise from hardware drivers (e.g., disk I/O, network interfaces) enhance entropy quality.
- **Hardware RNGs**: When available, hardware sources like Intel's RDRAND or AMD's RNGs are integrated, but mixed with software entropy to mitigate risks of bias or backdoors.

The kernel tracks entropy levels with an estimation counter, using cryptographic algorithms like ChaCha20 to stretch entropy into secure pseudorandom streams.

Key APIs

Linux offers several interfaces for random number access:

- **getrandom()**: Introduced in Linux 3.17, this system call is the modern, preferred method for obtaining secure random bytes directly from the kernel. It supports both blocking and non-blocking modes and avoids file descriptor overhead, making it ideal for cryptographic applications.
- **rand() and random()**: These C library functions rely on linear congruential generators (LCGs) and are not cryptographically secure, rendering them unsuitable for security-sensitive tasks.
- **/dev/random and /dev/urandom**: These legacy device files remain widely used for compatibility and low-level operations.

rngd Daemon and Hardware Integration

The rngd daemon, part of the rng-tools package, enhances entropy collection by:

- Retrieving data from hardware RNGs.
- Validating and conditioning the data (e.g., via FIPS 140-2 tests).
- Injecting high-quality entropy into the kernel's entropy pool through /dev/random.

This ensures hardware entropy is carefully integrated, reducing reliance on potentially untrustworthy sources.

In summary, Linux's RNG architecture combines diverse entropy sources, modern cryptographic algorithms like ChaCha20, and flexible APIs to deliver secure, reliable randomness for a wide range of applications.

3.3. RNG in Apple's macOS and iOS

Apple's macOS and iOS operating systems deliver a robust, hardware-accelerated random number generation (RNG) framework optimized for cryptographic security, performance, and adherence to industry standards. By combining system calls, hardware modules, and secure co-processors, Apple ensures high-quality entropy and cryptographically secure pseudorandom numbers.

Architecture Overview

The primary interface for secure randomness is the SecRandomCopyBytes() function within the Security framework, enabling applications to access cryptographically secure random bytes without directly interacting with low-level interfaces like /dev/random. While /dev/random is available, it is rarely used directly by developers, unlike in Linux. Apple's RNG is deeply integrated into the kernel and dedicated hardware, providing multiple layers of entropy sourcing and random data generation.

Entropy Sources

Apple collects entropy from both software and hardware sources to ensure unpredictability and resilience against entropy depletion:

- **Secure Enclave**: A dedicated co-processor with its own entropy source and RNG, used for cryptographic tasks like key management and encryption, operating in a secure, isolated environment.
- **T2 Security Chip** (on supported devices): Features a hardware RNG and cryptographic engine, isolated from the main CPU to protect against tampering and side-channel attacks.

- **System Events**: Software-based entropy is derived from device activities, such as I/O timings, interrupts, and user interactions.

This multi-source approach guarantees robust entropy, particularly during early boot or high-demand scenarios.

Cryptographic Expansion and DRBG Design

Apple's Deterministic Random Bit Generators (DRBGs) are designed for efficiency and security, likely aligning with NIST SP 800-90A standards. Key features include:

- **AES-based DRBG**: Used in FIPS-compliant environments, employing AES in Counter (CTR) mode to generate pseudorandom sequences from entropy.
- **ChaCha20-based Expansion**: Newer implementations leverage the ChaCha20 stream cipher for its speed, security, and resistance to timing attacks, ideal for mobile and energy-efficient devices.
- **Periodic Reseeding**: Regular reseeding with fresh hardware entropy ensures forward and backward secrecy, preventing prediction of output streams.
- **Sandbox Isolation**: RNG API access is restricted by application permissions, minimizing risks of misuse or exposure in cryptographic operations.

Security and Compliance

Apple's RNG systems are built to meet stringent cryptographic standards and support secure workflows, including biometric authentication, file system encryption, and encrypted messaging. Key aspects include:

- **Secure Enclave RNG**: Independently supports Touch ID, Face ID, Apple Pay, and secure key storage.
- **Secure Boot and FileVault**: Leverages RNG for boot processes and disk encryption.
- **Reseeding and Entropy Management**: Ensures resistance to entropy reuse and replay attacks through frequent reseeding and high entropy availability.

In summary, Apple's RNG architecture combines hardware-accelerated entropy, advanced cryptographic algorithms like ChaCha20, and strict security measures to deliver reliable, high-quality randomness for secure computing across macOS and iOS.

Random number generation (RNG) is critical for securing modern computing systems, underpinning encryption, authentication, secure communications, and digital signatures. The quality of randomness directly impacts data integrity and confidentiality. This analysis highlights the distinct yet robust RNG architectures in Microsoft Windows, Linux, and Apple's macOS/iOS, each tailored to meet cryptographic demands.

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.

Shared Principles and Challenges

All three operating systems prioritize:

- **Secure Entropy Collection**: Leveraging diverse hardware and software sources to ensure unpredictability.
- **Cryptographic Expansion**: Using standards-compliant DRBGs (e.g., NIST SP 800-90A) for robust pseudorandom output.
- **Periodic Reseeding**: Maintaining forward and backward secrecy to protect against state compromise.
- **Accessible APIs**: Enabling developers to integrate secure randomness into applications.

However, challenges persist, particularly in:

- **Early Boot Phases**: Limited entropy availability during system startup.
- **Embedded Systems**: Constrained hardware may struggle to provide sufficient entropy.
- **Heterogeneous Hardware**: Ensuring consistent entropy quality across diverse devices.

Future Directions

As cryptographic threats grow more sophisticated, operating systems must evolve their RNG systems by:

- Adopting **quantum-resistant algorithms** to counter emerging computational threats.
- Enhancing **entropy validation** to detect and mitigate degradation or manipulation.
- Promoting **transparency** through third-party audits and open-source contributions.

4. Conclusions

The strength of cryptographic systems rests on the quality of randomness. Windows, Linux, and Apple each implement sophisticated RNG architectures tailored to their ecosystems, yet they share a commitment to secure entropy, robust expansion, and developer accessibility. Continued advancements in RNG design will be essential to safeguarding modern computing against evolving threats.

References:
Используемая литература:
Foydalanilgan adabiyotlar:

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