American Journal of Applied Science and Technology

VOLUME Vol.05 Issue 05 2025
PAGE NO. 74-81
DOI 10.37547/ajast/Volume05Issue05-17

Random Number Generation in Operating Systems

Karimov Madjit Malikovich

Agency for Assessment of knowledge and competences under the ministry of Higher Education, Science and Innovation of the Republic of Uzbekistan, Tashkent, Uzbekistan

Komil Tashev

Department of Cryptology, Tashkent University of Information Technologies named after Muhammad al-Khwarizmi, Tashkent, Uzbekistan

Nuriddin Safoev

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

Tashmatova Shaxnoza Sabirovna

Tashkent State Technical University named after Islam Karimov, Tashkent, Uzbekistan

Qurbonova Kabira Erkinovna

Tashkent State Technical University named after Islam Karimov, Tashkent, Uzbekistan

Fayziraxmonov Boburjon Baxtiyorjon o‘g‘li

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

Received: 23 March 2025; **Accepted:** 19 April 2025; **Published:** 21 May 2025

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.

Introduction:

Random numbers are fundamental to numerous areas of computing, including simulations, gaming, randomized algorithms, and, most critically, security and cryptography. In this context, randomness is not merely a matter of variability; it is a foundational element that underpins the unpredictability and strength of cryptographic operations. Secure cryptographic systems rely on high-quality random numbers to ensure that data remains confidential, integrity is preserved, and attackers are unable to predict or reproduce security-critical values [1].

The principle “randomness equals security” is

especially true in cryptography, where various components are designed to be resistant to analysis and guessing [2]. For instance:

- **Encryption Keys** are generated using random data to ensure that no two keys are alike and that adversaries cannot derive them through any logical pattern. Predictable keys compromise the confidentiality of encrypted information.
- **Nonces** (numbers used once) are used in encryption schemes and secure communications to guarantee uniqueness across operations. If reused or predictable, they can enable replay attacks or compromise the integrity of secure channels.
- **Salts** are random values added to passwords before hashing to defend against precomputed attacks such as rainbow tables. Without salts, attackers could reverse-engineer hash values using known dictionaries of common passwords.
- **Secure Tokens**, used in session management and password resets, must be generated with high entropy to prevent session hijacking or unauthorized access through token prediction.

When randomness is weak or flawed, the entire security architecture becomes vulnerable. This has real-world implications: adversaries may be able to guess cryptographic keys, reproduce secure tokens, or even impersonate users in encrypted sessions [3]. The need for robust and unpredictable random numbers is therefore non-negotiable in the design of secure systems.

Major security protocols such as:

- **TLS** (Transport Layer Security), which secures web traffic (e.g., HTTPS)
- **VPNs** (Virtual Private Networks), which encrypt internet communication
- **Digital Signatures**, which verify the authenticity and integrity of data

all critically rely on random number generation to protect against attacks. In these protocols, randomness is used during key exchange, digital signature generation, and session establishment. For example, the TLS handshake process involves the exchange of random values to derive session keys. If these values can be guessed, an attacker could decrypt the supposedly secure communication.

To fulfill this essential role, modern operating systems (OSes) such as Microsoft Windows, Linux, and macOS embed RNGs at the core of their architecture. These RNGs are designed to gather entropy from a variety of unpredictable system-level sources, such as:

- User input events (e.g., keyboard and mouse movements)
- Timing variations in hardware activity (e.g., disk access or network latency)
- Hardware-based noise generators (e.g., Intel's RDRAND or TPM modules)

This entropy is then processed using cryptographically secure algorithms like AES in Counter (CTR) mode, ChaCha20, or SHA-based constructions, which expand limited entropy into large streams of high-quality pseudorandom bits.

To ensure ease of use and standardization, operating systems expose APIs through which applications and developers can request random data. These APIs ensure that developers do not need to implement their own randomness logic—a task prone to critical errors [4]. For instance, Windows provides BCryptGenRandom as part of its Cryptography API: Next Generation (CNG); Linux offers /dev/random, /dev/urandom, and the getrandom() syscall; macOS includes functions such as arc4random() and SecRandomCopyBytes().

While the core objective of these RNGs is consistent across platforms—to provide secure, high-quality randomness—their architectural designs and cryptographic strategies vary significantly, influenced by:

- **Platform Architecture**: Windows uses the CNG framework, Linux relies on device files and syscalls, and macOS integrates RNGs into system libraries like libSystem.
- **Entropy Collection Strategies**: Some platforms aggressively incorporate entropy from hardware RNGs, while others depend more heavily on event-based or software sources.
- **Deterministic Random Bit Generator (DRBG) Standards**: Many RNG implementations conform to NIST SP 800-90A standards, which define DRBGs based on AES, SHA-256, or HMAC constructions. However, some modern systems adopt stream ciphers like ChaCha20 for improved speed and security, particularly on resource-constrained or mobile platforms.

This paper surveys the design and implementation of random number generation in major operating systems, highlighting the architectural differences, entropy-handling mechanisms, DRBG standards, and APIs used to facilitate secure randomness. By analyzing these systems, we aim to identify best practices, potential weaknesses, and future directions for enhancing randomness in computing platforms.

Background and Importance

Random number generation (RNG) plays a foundational role in modern computing, especially in the field of cryptography, where the security of many algorithms and protocols hinges on the quality of randomness. Random numbers are crucial for generating encryption keys, initialization vectors, nonces, salts, secure session identifiers, and digital signatures. As such, the strength of encryption and the robustness of many security mechanisms are directly tied to the unpredictability and entropy of the underlying random number generation processes. An insecure RNG can lead to predictable outputs that undermine the entire security model of cryptographic systems [5].

The dangers of flawed RNGs have been demonstrated by real-world security incidents. One of the most notable examples is the Debian OpenSSL vulnerability that occurred between 2006 and 2008. In this case, a Debian developer mistakenly removed critical entropy-gathering code from the OpenSSL package, significantly weakening the randomness used to generate cryptographic keys. As a result, the affected systems produced a very limited number of possible keys—only 32,768 variations—making them highly susceptible to brute-force attacks. This vulnerability had far-reaching consequences, affecting SSH keys, SSL certificates, and other cryptographic elements across thousands of systems. It forced administrators worldwide to regenerate keys and re-secure their infrastructures, underscoring the catastrophic implications of insecure RNG implementations.

To prevent such failures, modern operating systems and cryptographic libraries incorporate robust RNG designs composed of several interdependent components [6]. These typically include:

- **Entropy Sources**: These are raw, unpredictable inputs collected from system behavior or dedicated hardware. Examples include timing variations in keystrokes, mouse movements, disk I/O latencies, and hardware-based noise generators (e.g., Intel's RDRAND, AMD's RdSeed, or physical entropy devices like TPMs and HWRNGs). The quality of these sources is essential, as they form the basis of the system's randomness.
- **Entropy Pool**: This is a system-managed structure that aggregates the entropy gathered from various sources. The pool acts as a buffer, storing collected randomness until there is enough to securely seed a deterministic generator. Some systems maintain multiple pools for different purposes, and entropy accounting mechanisms are used to estimate how much unpredictability has been accumulated. For example, Linux uses separate pools for /dev/random and /dev/urandom, with blocking behavior depending on entropy availability.
- **Pseudorandom Number Generators (PRGs) or Deterministic Random Bit Generators (DRBGs):** These are cryptographic algorithms designed to expand a limited amount of high-quality entropy into a long stream of random bits. Once seeded, these generators can rapidly produce large amounts of pseudorandom data. Common DRBGs are based on algorithms such as AES in counter mode (AES-CTR DRBG), HMAC constructions (HMAC DRBG), and secure hash functions (Hash DRBG). More recently, stream ciphers like ChaCha20 have also been adopted in RNG designs for both speed and security. These generators are typically compliant with NIST Special Publication 800-90A/B/C standards and are regularly reviewed for cryptographic soundness.
- **Application Programming Interfaces (APIs):** To ensure usability and consistency, operating systems provide standardized APIs for developers to access random data. These APIs abstract away the internal complexity and guarantee secure outputs if used correctly. For example, Windows offers interfaces like BCryptGenRandom and RtlGenRandom; Linux exposes randomness through devices such as /dev/random, /dev/urandom, and the getrandom() syscall; macOS provides interfaces like SecRandomCopyBytes and arc4random(). The proper use of these APIs is essential, as bypassing them or misusing insecure alternatives (such as using non-cryptographic RNGs like rand() or srand() in C) can introduce subtle but dangerous vulnerabilities.

Overall, the integrity of random number generation systems is a critical pillar of digital security. A failure at this level can result in broken encryption, impersonation attacks, token prediction, or unauthorized data access. Hence, RNGs must be implemented, audited, and maintained with the highest standards of cryptographic rigor. As threats evolve and new vulnerabilities are discovered, operating system vendors continue to enhance the design and performance of their RNG subsystems, ensuring that the cryptographic primitives depending on them remain robust and trustworthy.

Random Number Generation in Operating Systems

- **RNG in Microsoft Windows**

Microsoft Windows employs a robust cryptographic framework known as Cryptography API: Next Generation (CNG) to handle cryptographic services, including random number generation. At the core of Windows' RNG system lies an implementation of a Deterministic Random Bit Generator (DRBG) based on the AES block cipher in Counter (CTR) mode, which is

compliant with the NIST SP 800-90A standard. This ensures both compliance with federal cryptographic guidelines and a high level of cryptographic strength suitable for secure communications and system-level operations [7].

Entropy Sources

Windows collects entropy from multiple hardware and software sources to populate its internal entropy pool. The diversity of these sources helps enhance the unpredictability and resistance against entropy exhaustion or manipulation:

- **Hardware Random Number Generators (HRNGs):** Windows utilizes hardware RNGs when available, such as Intel's RDRAND instruction, which provides high-speed, hardware-generated entropy.
- **Timing of System Events**: The system records time variations in low-level events, such as interrupts and system calls, to capture unpredictability arising from user and system activity.
- **User Input Devices**: Timing data from mouse movements and keystrokes serves as a source of entropy, especially during system startup or before sufficient entropy has accumulated.
- **Network Activity**: Variations in network packet arrival times and traffic behavior contribute to randomness.
- **Disk Operations**: Similar to network timing, read/write latencies on storage devices are used to extract additional entropy.

This multi-source approach ensures that Windows can gather sufficient entropy across a wide range of operational contexts, including headless servers or unattended systems.

Key RNG Components and APIs

Windows exposes several components and interfaces for random number generation, targeted at both system-level functions and application developers:

- **SystemPrng**: This is the core pseudorandom generator used internally by the Windows kernel. It is responsible for generating cryptographic-quality random data for system components and reseeding itself periodically to maintain security properties like forward and backward secrecy.
- **BCryptGenRandom():** The primary public-facing API for developers to access random data. This function allows both user-mode and kernel-mode applications to retrieve cryptographically secure random bytes. It supports two modes:

o Using the system-preferred RNG (BCRYPT_USE_SYSTEM_PREFERRED_RNG flag)

o Specifying a custom algorithm provider (e.g., a hardware-based RNG)

BCryptGenRandom() is widely adopted within the Windows ecosystem and is suitable for generating encryption keys, session tokens, nonces, salts, and other sensitive values.

Security Measures and Reseeding

Windows CNG is designed with strong security guarantees:

- **Forward Secrecy**: If the RNG state is compromised at time t, it should not reveal any information about outputs generated before t. This is achieved by frequent reseeding with fresh entropy.
- **Backward Secrecy**: If the RNG state is compromised at time t, it should not compromise outputs generated after t, as new entropy is injected periodically.
- **Self-Healing and Monitoring**: The DRBG monitors entropy health and triggers reseeding or failure if entropy sources degrade. This helps defend against entropy starvation or deliberate manipulation.

In summary, Windows provides a well-architected RNG system integrated with its cryptographic infrastructure. By combining hardware entropy, system noise, and strict adherence to NIST standards, it ensures a high degree of randomness suitable for secure computing.

o **RNG in Linux Systems**

Linux systems implement a well-structured and evolving architecture for random number generation that is integral to both system security and cryptographic operations [8-10]. The Linux RNG architecture is designed to collect entropy from diverse sources and deliver secure pseudorandom data to both kernel and user-space applications. Its design balances performance, reliability, and cryptographic strength.

Architecture Overview

The Linux kernel employs a dual-RNG interface, historically centered around two special device files: /dev/random and /dev/urandom. Each serves different purposes and has distinct behavior regarding entropy availability:

- **/dev/random**: This interface is blocking, meaning it only returns data when sufficient entropy is available in the internal entropy pool. It is designed for applications that demand very high-quality randomness, such as key generation in cryptographic tools.
- **/dev/urandom**: This interface is non-blocking

and returns random bytes even if the entropy pool is low. It uses a cryptographically secure pseudorandom number generator (CSPRNG) to stretch available entropy. While historically viewed as slightly less secure, modern cryptographic implementations have largely closed this gap.

Starting from Linux kernel 5.6, the internal RNG has been upgraded to use a ChaCha20-based DRBG, replacing the older SHA-1 based algorithm. ChaCha20 offers excellent performance and is resistant to known cryptanalytic attacks, making it a modern and efficient choice for both user and kernel applications.

Entropy Collection

Entropy in Linux is harvested from a wide array of system activities, which contribute to the internal entropy pool:

- **Interrupt Timings**: The kernel records the precise timing of interrupts, which are naturally asynchronous and unpredictable, making them a good entropy source.
- **Keyboard and Mouse Events**: User input events, such as keypress timings and mouse movements, are collected, especially during system boot or early startup.
- **Device Drivers**: Many hardware drivers (e.g., disk I/O, network interfaces) provide timing information and low-level noise that is used to improve entropy quality.
- **Hardware RNGs**: If present, hardware-based entropy sources (e.g., Intel's RDRAND or AMD's hardware RNGs) are integrated into the entropy pool. However, hardware entropy is typically mixed with software sources to reduce the risk of backdoors or biases.

Entropy is tracked using an entropy estimation counter, and entropy is periodically extracted and stretched into longer pseudorandom streams using cryptographic algorithms like ChaCha20.

Key APIs

Linux provides several interfaces for accessing random numbers:

- **getrandom()**: Introduced in Linux 3.17, getrandom() is the preferred modern system call for accessing secure random bytes directly from the kernel. It avoids file descriptor usage and provides both blocking and non-blocking options. It is the recommended choice for cryptographic applications and system-level security.
- **rand() and random():** These are standard C library functions but are not cryptographically secure. They are based on linear congruential generators (LCGs) and are considered insecure for any cryptographic purpose. Their use is strongly discouraged in security-sensitive contexts.
- **/dev/random and /dev/urandom**: Still widely used in legacy and portable applications, these devices remain essential interfaces for backward compatibility and low-level operations.

rngd Daemon and Hardware Integration

To support hardware entropy devices and ensure proper mixing of entropy into the system RNG, Linux supports the rngd daemon (part of the rng-tools package). This daemon:

- Reads entropy from hardware RNGs.
- Validates and conditions the data (e.g., using FIPS 140-2 tests).
- Injects high-quality entropy into the kernel's entropy pool via /dev/random.

This process ensures that hardware entropy so urces are not blindly trusted and that their output is carefully integrated into the system.

- **RNG in Apple's macOS and iOS**

Apple's operating systems—macOS and iOS—implement a tightly integrated and hardware-accelerated approach to random number generation, focusing on cryptographic security, performance, and compliance with industry standards. These RNG systems leverage a combination of system calls, hardware modules, and secure co-processors to generate high-quality entropy and cryptographically strong pseudorandom numbers [11].

Architecture Overview

The primary interface for accessing secure randomness in Apple platforms is the SecRandomCopyBytes() function, part of the Security framework. This high-level API allows applications to request cryptographically secure random bytes without having to directly access lower-level interfaces or device files.

Internally, SecRandomCopyBytes() interacts with system RNG components that may also access /dev/random. However, in contrast to Linux, these lower-level interfaces are not typically used directly by application developers on Apple platforms.

Apple's architecture integrates RNG functionality deeply into both the kernel space and dedicated hardware modules, offering multiple layers of entropy sourcing and random data expansion.

Entropy Sources

Entropy in macOS and iOS is collected from both software-based system activity and dedicated

hardware components, including:

- **Apple Secure Enclave**: A separate co-processor with its own entropy source, random number generator, and secure execution environment. It is used extensively in cryptographic operations, including key management and encryption tasks.
- **Apple T2 Security Chip (on supported devices):** Includes a hardware random number generator and cryptographic engine that enhances entropy generation and security. The T2 chip operates in isolation from the main CPU to prevent tampering and side-channel attacks.
- **System Events**: Software-level entropy from device activity such as I/O timings, interrupts, and user interaction is also included.

This multi-source entropy collection strategy ensures robust unpredictability and resistance to entropy starvation, especially during early boot or high-demand periods.

Cryptographic Expansion and DRBG Design

Apple employs hardware-assisted DRBGs (Deterministic Random Bit Generators), which are likely compliant with NIST SP 800-90A standards [12-18]. The following features characterize their DRBG implementations:

- **AES-based DRBG**: For environments favoring FIPS compliance, Apple devices utilize AES in counter (CTR) mode to expand entropy into pseudorandom sequences.
- **ChaCha20-based Expansion**: Newer Apple RNG implementations have incorporated ChaCha20, a stream cipher favored for its speed, security, and resistance to timing attacks. It is especially effective on mobile hardware and energy-efficient processors.
- **Periodic Reseeding**: RNG instances in macOS and iOS are regularly reseeded with fresh entropy from hardware sources, preventing both forward and backward prediction of the output stream.
- **Sandbox Isolation:** Access to the RNG APIs is sandboxed and restricted per application permissions. This security model reduces the risk of RNG misuse or unintended exposure of cryptographic operations.

Security and Compliance

Apple RNG systems are designed to meet high standards of cryptographic assurance and are integrated into secure workflows involving biometric authentication, file system encryption, and encrypted messaging. Notably:

- The Secure Enclave maintains its own independent RNG to support Touch ID, Face ID, Apple Pay, and secure key storage.
- Secure Boot and FileVault encryption routines utilize the RNG system during boot and disk decryption phases.
- Reseeding mechanisms, along with high entropy availability, ensure resistance to entropy reuse and replay vulnerabilities.

Table 1. Comparative Analysis.

Feature	Windows	Linux	macOS/iOS
DRBG Type	AES-CTR (CNG)	ChaCha20 (since kernel 5.6)	AES / ChaCha20
Secure API	`BCryptGenRandom()`	`getrandom()`, `/dev/urandom`	`SecRandomCopyBytes()`
Hardware RNG Support	RDRAND, TPM	RDRAND, RNGd, TPM	T2 chip, Secure Enclave
Entropy Pool	Internal pool managed by kernel	Linux entropy pool	Apple entropy manager
Blocking API	No	`/dev/random` (blocking)	No

In Table 1, we present a comparative analysis of the random number generation mechanisms implemented across three major operating systems: Windows, Linux, and macOS/iOS. Each platform employs different deterministic random bit generators (DRBGs), entropy collection strategies, and APIs to provide secure randomness. Windows relies on the Cryptography API: Next Generation (CNG) framework, utilizing an AES-CTR-based DRBG that conforms to NIST SP 800-90A standards. Linux, on the other hand, transitioned to a ChaCha20-based DRBG in kernel version 5.6 and offers both blocking and non-blocking interfaces for random number access. macOS and iOS integrate hardware-assisted DRBGs, drawing entropy from dedicated hardware components like the Secure Enclave and T2 chip, and use both AES and ChaCha20 for expansion. All three operating systems support secure APIs for application-level randomness, although they differ in how they expose blocking behavior and manage entropy pools internally. This comparison highlights the diverse approaches taken by modern operating

systems to ensure cryptographic security through robust random number generation.

CONCLUSIONS

Random number generation is a foundational element in securing modern computing systems. From encryption and authentication to secure communications and digital signatures, the quality of randomness directly affects the integrity and confidentiality of data. As this survey has shown, Microsoft Windows, Linux, and Apple's macOS/iOS each implement distinct yet robust RNG architectures to meet cryptographic needs.

Windows relies on its CNG framework and AES-CTR DRBG with structured entropy inputs from hardware and system events. Linux adopts a dual-interface model with /dev/random, /dev/urandom, and getrandom(), using ChaCha20-based DRBGs and extensive kernel entropy sources. Apple platforms integrate RNG tightly with dedicated hardware such as the Secure Enclave and T2 chip, favoring both AES and ChaCha20 algorithms for high-assurance entropy expansion.

Despite these differences, all major OSes emphasize several core principles: secure entropy collection, cryptographic expansion, periodic reseeding, and accessible APIs for developers. However, challenges remain—especially in early boot phases, embedded systems, and ensuring consistent entropy quality across heterogeneous hardware.

As cryptographic threats evolve and attack vectors become more sophisticated, operating systems must continue advancing their RNG mechanisms. This includes integrating quantum-resistant algorithms, improving entropy validation, and ensuring transparency through third-party audits and open-source contributions. Ultimately, the strength of any security system hinges on its foundation—and in cryptography, that foundation begins with randomness.

REFERENCES

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

Eastlake, D., Schiller, J., & Crocker, S. (2005). Randomness Requirements for Security. RFC 4086. https://www.rfc-editor.org/rfc/rfc4086

Microsoft. (2023). Cryptography API: Next Generation. Microsoft Docs. https://learn.microsoft.com/en-us/windows/win32/seccng/cng-portal

Microsoft. (2023). BCryptGenRandom function (bcrypt.h). Microsoft Docs. https://learn.microsoft.com/en-us/windows/win32/api/bcrypt/nf-bcrypt-bcryptgenrandom

Linux Kernel Documentation. (2023). Random Number Generator. https://www.kernel.org/doc/html/latest/admin-guide/dev-random.html

Linux man-pages project. (2023). getrandom(2) – Linux manual page. https://man7.org/linux/man-pages/man2/getrandom.2.html

Apple Developer Documentation. (2023). SecRandomCopyBytes. https://developer.apple.com/documentation/security/1399291-secrandomcopybytes

Apple. (2020). Platform Security Guide. https://support.apple.com/guide/security/welcome/web

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

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

Lacharme, P. (2012). Security flaws in Linux's /dev/random. https://eprint.iacr.org/2012/251

BSD Unix. (2022). arc4random and related APIs. https://man.openbsd.org/arc4random

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

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

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

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

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

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