Wikipedia
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
العربية
Català
Čeština
Español
فارسی
Français
한국어
Italiano
עברית
Nederlands
日本語
Polski
Português
Русский
Slovenščina
Suomi
Svenska
ไทย
Türkçe
Українська
اردو
Edit links

Pseudorandom number generator

From Wikipedia, the free encyclopedia

This article **has an unclear citation style**. The references used may be made clearer with a different or consistent style of citation, footnoting, or external linking. *(September 2009)*

A **pseudorandom number generator** (**PRNG**), also known as a **deterministic random bit generator** (**DRBG**),[1] is an algorithm for generating a sequence of numbers whose properties approximate the properties of sequences of random numbers. The PRNG-generated sequence is not truly random, because it is completely determined by a relatively small set of initial values, called the PRNG's *seed* (which may include truly random values). Although sequences that are closer to truly random can be generated using hardware random number generators, *pseudorandom* number generators are important in practice for their speed in number generation and their reproducibility.[2]

PRNGs are central in applications such as simulations (e.g. for the Monte Carlo method), electronic games (e.g. for procedural generation), and cryptography. Cryptographic applications require the output not to be predictable from earlier outputs, and more elaborate algorithms, which do not inherit the linearity of simpler PRNGs, are needed.

Good statistical properties are a central requirement for the output of a PRNG. In general, careful mathematical analysis is required to have any confidence that a PRNG generates numbers that are sufficiently close to random to suit the intended use. John von Neumann cautioned about the misinterpretation of a PRNG as a truly random generator, and joked that "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."[3]

Contents [hide]

Periodicity [edit]

A PRNG can be started from an arbitrary initial state using a seed state. It will always produce the same sequence when initialized with that state. The *period* of a PRNG is defined thus: the maximum, over all starting states, of the length of the repetition-free prefix of the sequence. The period is bounded by the number of the states, usually measured in bits. However, since the length of the period potentially doubles with each bit of "state" added, it is easy to build PRNGs with periods long enough for many practical applications.

If a PRNG's internal state contains *n* bits, its period can be no longer than 2^n results, and may be much shorter. For some PRNGs, the period length can be calculated without walking through the whole period. Linear Feedback Shift Registers (LFSRs) are usually chosen to have periods of exactly $2^n - 1$. Linear congruential generators have periods that can be calculated by factoring.[*citation needed*] Although PRNGs will repeat their results after they reach the end of their period, a repeated result does not imply that the end of the period has been reached, since its internal state may be larger than its output; this is particularly obvious with PRNGs with a one-bit output.

Most PRNG algorithms produce sequences which are uniformly distributed by any of several tests. It is an open question, and one central to the theory and practice of cryptography, whether there is any way to distinguish the output of a high-quality PRNG from a truly random sequence without knowing the algorithm(s) used and the state with which it was initialized. The security of most cryptographic algorithms and protocols using PRNGs is based on the assumption that it is infeasible to distinguish use of a suitable PRNG from use of a truly random sequence. The simplest examples of this dependency are stream ciphers, which (most often) work by exclusive or-ing the plaintext of a message with the output of a PRNG, producing ciphertext. The design of cryptographically adequate PRNGs is extremely difficult, because they must meet additional criteria (see below). The size of its period is an important factor in the cryptographic suitability of a PRNG, but not the only one.

Potential problems with deterministic generators [edit]

In practice, the output from many common PRNGs exhibit artifacts that cause them to fail statistical pattern-detection tests. These include:

- Shorter than expected periods for some seed states (such seed states may be called 'weak' in this context);
- Lack of uniformity of distribution for large numbers of generated numbers;
- Correlation of successive values;
- Poor dimensional distribution of the output sequence;
- The distances between where certain values occur are distributed differently from those in a random sequence distribution.

Defects exhibited by flawed PRNGs range from unnoticeable (and unknown) to very obvious. An example was the RANDU random number algorithm used for decades on mainframe computers. It was seriously flawed, but its inadequacy went undetected for a very long time.

In many fields, much research work prior to the 21st century that relied on random selection or on Monte Carlo simulations, or in other ways relied on PRNGs, is much less reliable than it might have been as a result of using poor-quality PRNGs.[4] Even today, caution is sometimes required, as illustrated by the following warning, which is given in the *International Encyclopedia of Statistical Science* (2010).[5]

> The list of widely used generators that should be discarded is [long] ... Check the default [PRNG] of your favorite software and be ready to replace it if needed. This last recommendation has been made over and over again over the past 40 years. Perhaps amazingly, it remains as relevant today as it was 40 years ago.

As an illustration, consider the widely used programming language Java. As of 2015, Java still relies on a linear congruential generator (LCG) for its (non-cryptographically-secure) PRNG;[6] yet LCGs are of low quality—see further below.

The first PRNG to avoid major problems and still run fairly quickly was the Mersenne Twister (discussed below), which was published in 1998. Other high-quality PRNGs have since been developed.

Generators based on linear recurrences [edit]

In the second half of the 20th century, the standard class of algorithms used for PRNGs comprised linear congruential generators. The quality of LCGs was known to be inadequate, but better methods were unavailable. Press et al. (2007) described the result thus: "If all scientific papers whose results are in doubt because of [LCGs and related] were to disappear from library shelves, there would be a gap on each shelf about as big as your fist".[7]

A major advance in the construction of pseudorandom generators was the introduction of techniques based on **linear recurrences** on the two-element field; such generators are related to linear feedback shift registers.

The 1997 invention of the Mersenne twister,[8] in particular, avoided many of the problems with earlier generators. The Mersenne Twister has a period of $2^{19937}-1$ iterations ($\approx 4.3 \times 10^{6001}$), is proven to be equidistributed in (up to) 623 dimensions (for 32-bit values), and at the time of its introduction was running faster than other statistically reasonable generators.

Subsequently, the WELL family of generators was developed.[9] The WELL generators in some ways improves on the quality of the Mersenne Twister—which has a too-large state space and a very slow recovery from state spaces with a large number of zeros.

In 2003, George Marsaglia introduced the family of xorshift generators,[10] again based on a linear recurrence. Such generators are extremely fast and, combined with a nonlinear operation, they pass strong statistical tests.[11]

Cryptographically secure pseudorandom number generators [edit]

Main article: Cryptographically secure pseudorandom number generator

A PRNG suitable for cryptographic applications is called a *cryptographically secure PRNG* (CSPRNG). A requirement for a CSPRNG is that an adversary not knowing the seed has only negligible advantage in distinguishing the generator's output sequence from a random sequence. In other words, while a PRNG is only required to pass certain statistical tests, a CSPRNG must pass all statistical tests that are restricted to polynomial time in the size of the seed. Though such property cannot be proven, strong evidence may be provided by reducing the CSPRNG to a problem that is assumed to be hard, such as integer factorization.[12] In general, years of review may be required before an algorithm can be certified as a CSPRNG.

Some classes of CSPRNGs include the following:

- Stream ciphers
- Block ciphers running in counter[13] or output feedback mode.
- PRNGs that have been designed specifically to be cryptographically secure, such as Microsoft's Cryptographic Application Programming Interface function CryptGenRandom, the Yarrow algorithm (incorporated in Mac OS X and FreeBSD), and Fortuna.
- Combination PRNGs which attempt to combine several PRNG primitive algorithms with the goal of removing any non-randomness.
- Special designs based on mathematical hardness assumptions. Examples include Micali-Schnorr and the Blum Blum Shub algorithm, which provide a strong security proof. Such algorithms are rather slow compared to traditional constructions, and impractical for many applications.

It has been shown to be likely that the NSA has inserted an asymmetric backdoor into the NIST certified pseudorandom number generator Dual_EC_DRBG.[14]

BSI evaluation criteria [edit]

The German Federal Office for Information Security (*Bundesamt für Sicherheit in der Informationstechnik*, BSI) has established four criteria for quality of deterministic random number generators.[15] They are summarized here:

- K1 — A sequence of random numbers with a low probability of containing identical consecutive elements.
- K2 — A sequence of numbers which is indistinguishable from 'true random' numbers according to specified statistical tests. The tests are the *monobit* test (equal numbers of ones and zeros in the sequence), *poker* test (a special instance of the chi-squared test), *runs* test (counts the frequency of runs of various lengths), *longruns* test (checks whether there exists any run of length 34 or greater in 20 000 bits of the sequence) — both from BSI[15] and NIST,[16] and the *autocorrelation* test. In essence, these requirements are a test of how well a bit sequence: has zeros and ones equally often; after a sequence of *n* zeros (or ones), the next bit a one (or zero) with probability one-half; and any selected subsequence contains no information about the next element(s) in the sequence.
- K3 — It should be impossible for any attacker (for all practical purposes) to calculate, or otherwise guess, from any given sub-sequence, any previous or future values in the sequence, nor any inner state of the generator.
- K4 — It should be impossible, for all practical purposes, for an attacker to calculate, or guess from an inner state of the generator, any previous numbers in the sequence or any previous inner generator states.

For cryptographic applications, only generators meeting the K3 or K4 standard are acceptable.

Mathematical definition [edit]

Given

- P - a probability distribution on $\left(\mathbb{R}, \mathfrak{B}\right)$ (where $\mathfrak{B}$ is the standard Borel field on the real line)
- $\mathfrak{F}$ - a non-empty collection of Borel sets $\mathfrak{F} \subseteq \mathfrak{B}$, e.g. $\mathfrak{F} = \left\{\left(-\infty, t\right] : t \in \mathbb{R}\right\}$. If $\mathfrak{F}$ is not specified, it may be either $\mathfrak{B}$ or $\left\{\left(-\infty, t\right] : t \in \mathbb{R}\right\}$, depending on context.
- $A \subseteq \mathbb{R}$ - a non-empty set (not necessarily a Borel set). Often A is a set between P's support and its interior, for instance, if P is the uniform distribution on the interval $\left(0, 1\right]$, A might be $\left(0, 1\right]$. If A is not specified, it is assumed to be some set contained in the support of P and containing its interior, depending on context.

we call a function $f : \mathbb{N}_1 \rightarrow \mathbb{R}$ (where $\mathbb{N}_1 = \left\{1, 2, 3, \dots\right\}$ is the set of positive integers) a **pseudo-random number generator for P given $\mathfrak{F}$ taking values in A** iff

- $f\left(\mathbb{N}_1\right) \subseteq A$
- $$\forall E \in \mathfrak{F} \quad \forall 0 < \varepsilon \in \mathbb{R} \quad \exists N \in \mathbb{N}_1 \quad \forall N \leq n \in \mathbb{N}_1, \quad \left|\frac{\#\left\{i \in \left\{1, 2, \dots, n\right\} : f(i) \in E\right\}}{n} - P(E)\right| < \varepsilon$$

($\#S$ denotes the number of elements in the finite set S.)

It can be shown that if f is a pseudo-random number generator for the uniform distribution on $\left(0, 1\right)$ and if F is the CDF of some given probability distribution P, then $F^* \circ f$ is a pseudo-random number generator for P, where $F^* : \left(0, 1\right) \rightarrow \mathbb{R}$ is the percentile of P, i.e. $F^*(x) := \inf\left\{t \in \mathbb{R} : x \leq F(t)\right\}$. Intuitively, an arbitrary distribution can be simulated from a simulation of the standard uniform distribution.

Early approaches [edit]

An early computer-based PRNG, suggested by John von Neumann in 1946, is known as the middle-square method. The algorithm is as follows: take any number, square it, remove the middle digits of the resulting number as the "random number", then use that number as the seed for the next iteration. For example, squaring the number "1111" yields "1234321", which can be written as "01234321", an 8-digit number being the square of a 4-digit number. This gives "2343" as the "random" number. Repeating this procedure gives "4896" as the next result, and so on. Von Neumann used 10 digit numbers, but the process was the same.

A problem with the "middle square" method is that all sequences eventually repeat themselves, some very quickly, such as "0000". Von Neumann was aware of this, but he found the approach sufficient for his purposes, and was worried that mathematical "fixes" would simply hide errors rather than remove them.

Von Neumann judged hardware random number generators unsuitable, for, if they did not record the output generated, they could not later be tested for errors. If they did record their output, they would exhaust the limited computer memories then available, and so the computer's ability to read and write numbers. If the numbers were written to cards, they would take very much longer to write and read. On the ENIAC computer he was using, the "middle square" method generated numbers at a rate some hundred times faster than reading numbers in from punched cards.

The middle-square method has since been supplanted by more elaborate generators.

Non-uniform generators [edit]

Main article: Pseudo-random number sampling

Numbers selected from a non-uniform probability distribution can be generated using a uniform distribution PRNG and a function that relates the two distributions.

First, one needs the cumulative distribution function $F(b)$ of the target distribution $f(b)$:

$$F(b) = \int_{-\infty}^{b} f(b')db'$$

Note that $0 = F(-\infty) \leq F(b) \leq F(\infty) = 1$. Using a random number *c* from a uniform distribution as the probability density to "pass by", we get

$$F(b) = c$$

so that

$$b = F^{-1}(c)$$

is a number randomly selected from distribution $f(b)$.

For example, the inverse of cumulative Gaussian distribution $\operatorname{erf}^{-1}(x)$ with an ideal uniform PRNG with range (0, 1) as input x would produce a sequence of (positive only) values with a Gaussian distribution; however

- when using practical number representations, the infinite "tails" of the distribution have to be truncated to finite values.
- Repetitive recalculation of $\operatorname{erf}^{-1}(x)$ should be reduced by means such as ziggurat algorithm for faster generation.

Similar considerations apply to generating other non-uniform distributions such as Rayleigh and Poisson.

See also [edit]

- List of pseudorandom number generators
- Applications of randomness
- Low-discrepancy sequence
- Pseudorandom binary sequence
- Pseudorandom noise
- Random number generation
- Random number generator attack
- Randomness
- Statistical randomness

Mathematics portal

References [edit]

1. ^ Barker, Elaine; Barker, William; Burr, William; Polk, William; Smid, Miles (July 2012). "Recommendation for Key Management" (PDF). *NIST Special Publication 800-57*. NIST. Retrieved 19 August 2013.
2. ^ "Learn about pseudo random number generators". *https://www.khanacademy.orG*.
3. ^ Von Neumann, John (1951). "Various techniques used in connection with random digits" (PDF). *National Bureau of Standards Applied Mathematics Series* **12**: 36–38.
4. ^ Press et al. (2007), chap.7
5. ^ L'Ecuyer P. (2010), "Uniform random number generators", *International Encyclopedia of Statistical Science* (editor—Lovric M.) Springer.
6. ^ Random.java at OpenJDK.
7. ^ Press et al. (2007) §7.1
8. ^ Matsumoto, Makoto; Nishimura, Takuji (1998). "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". *ACM Transactions on Modeling and Computer Simulation* (ACM) **8** (1): 3–30. doi:10.1145/272991.272995.
9. ^ Panneton, François; L'Ecuyer, Pierre; Matsumoto, Makoto (2006). "Improved long-period generators based on linear recurrences modulo 2". *ACM Transactions on Mathematical Software* **32** (1): 1–16. doi:10.1145/1132973.1132974.
10. ^ Marsaglia, George (July 2003). "Xorshift RNGs". *Journal of Statistical Software* **8** (14).
11. ^ S.Vigna. "xorshift*/xorshift+ generators and the PRNG shootout".
12. ^ Song Y. Yan. *Cryptanalytic Attacks on RSA*. Springer, 2007. p. 73. ISBN 978-0-387-48741-0.
13. ^ Niels Ferguson, Bruce Schneier, Tadayoshi Kohno (2010). "Cryptography Engineering: Design Principles and Practical Applications, Chapter 9.4: The Generator" (PDF).
14. ^ Matthew Green. "The Many Flaws of Dual_EC_DRBG".
15. ^ [a] [b] Schindler, Werner (2 December 1999). "Functionality Classes and Evaluation Methodology for Deterministic Random Number Generators" (PDF). *Anwendungshinweise und Interpretationen (AIS)*. Bundesamt für Sicherheit in der Informationstechnik. pp. 5–11. Retrieved 19 August 2013.
16. ^ "Security requirements for cryptographic modules". *FIPS*. NIST. 1994-01-11. p. 4.11.1 Power–Up Tests. Retrieved 19 August 2013.

Bibliography [edit]

- Barker E., Kelsey J., *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, NIST SP800-90A, January 2012
- Brent R.P., "Some long-period random number generators using shifts and xors", *ANZIAM Journal*, 2007; 48:C188–C202
- Gentle J.E. (2003), *Random Number Generation and Monte Carlo Methods*, Springer.
- Hörmann W., Leydold J., Derflinger G. (2004, 2011), *Automatic Nonuniform Random Variate Generation*, Springer-Verlag.
- Knuth D.E.. *The Art of Computer Programming*, Volume 2: *Seminumerical Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89684-2. Chapter 3. [Extensive coverage of statistical tests for non-randomness.]
- Luby M., *Pseudorandomness and Cryptographic Applications*, Princeton Univ Press, 1996. ISBN 9780691025469
- Matthews R., "Maximally Periodic Reciprocals", *Bulletin of the Institute of Mathematics and its Applications*, 28: 147-148, 1992.
- von Neumann J., "Various techniques used in connection with random digits," in A.S. Householder, G.E. Forsythe, and H.H. Germond, eds., *Monte Carlo Method*, National Bureau of Standards Applied Mathematics Series, 12 (Washington, D.C.: U.S. Government Printing Office, 1951): 36-38.
- Peterson, Ivars (1997). *The Jungles of Randomness : a mathematical safari*. New York: John Wiley & Sons. ISBN 0-471-16449-6.
- Press W.H., Teukolsky S.A., Vetterling W.T., Flannery B.P. (2007), *Numerical Recipes* (Cambridge University Press).

- Viega J., "Practical Random Number Generation in Software", in Proc. 19th Annual Computer Security Applications Conference, Dec. 2003.

External links [edit]

- TestU01: A free, state-of-the-art (GPL) C++ Random Number Test Suite.
- DieHarder: A free (GPL) C Random Number Test Suite.
- "Generating random numbers" (in embedded systems) by Eric Uner (2004)
- "Analysis of the Linux Random Number Generator" by Zvi Gutterman, Benny Pinkas, and Tzachy Reinman (2006)
- "Better pseudorandom generators" by Parikshit Gopalan, Raghu Meka, Omer Reingold, Luca Trevisan, and Salil Vadhan (Microsoft Research, 2012)

Categories: Pseudorandom number generators