

Random number generator

A **random number generator** (often abbreviated as RNG) is a computational or physical device designed to generate a sequence of [numbers](#) or symbols that lack any pattern, i.e. appear [random](#). Computer-based systems for random number generation are widely used, but often fall short of this goal, though they may meet some statistical tests for randomness intended to ensure that they do not have any easily discernible patterns. Methods for generating random results have existed since ancient times, including [dice](#), [coin flipping](#), the [shuffling](#) of [playing cards](#), the use of yarrow stalks in the [I Ching](#), and many other techniques.

"True" random numbers vs. pseudo-random numbers

There are two principal methods used to generate random numbers. One measures some physical phenomenon that is expected to be random and then compensates for possible biases in the measurement process. The other uses computational [algorithms](#) that produce long sequences of apparently random results, which are in fact determined by a shorter initial seed or key. The latter type are often called [pseudo-random number generators](#).

A "random number generator" based solely on deterministic computation *cannot* be regarded as a "true" random number generator, since its output is inherently predictable. [John von Neumann](#) famously said "Anyone who uses arithmetic methods to produce random numbers is in a state of sin." How to distinguish a "true" random number from the output of a pseudo-random number generator is a very difficult problem. However, carefully chosen pseudo-random number generators can be used instead of true random numbers in many applications. Rigorous statistical analysis of the output is often needed to have confidence in the algorithm.

Random numbers in computing

Most computer programming languages include functions or library routines that purport to be random number generators. They are often designed to provide a random byte or word, or a [floating point](#) number [uniformly distributed](#) between 0 and 1.

Such library functions often have poor statistical properties and some will repeat patterns after only tens of thousands of trials. They are often initialized using a computer's [real time clock](#) as the seed. These functions may provide enough randomness for certain tasks (for example simple video games) but are unsuitable where high-quality randomness is required, such as in cryptographic applications, statistics or numerical analysis. Much higher quality random number sources are available on most operating systems; for example [/dev/random](#) on various BSD flavors, Linux, Mac OS X, IRIX, and Solaris, or [CryptGenRandom](#) for Microsoft Windows.

Generating random numbers from physical processes

There is general agreement that, if there are such things as "true" random numbers, they are most likely to be found by looking at physical processes which are, as far as we know, unpredictable.

A physical random number generator can be based on an essentially random atomic or subatomic physical phenomenon whose randomness can be traced to the laws of [quantum mechanics](#). An example of this is the [Atari](#) gaming console, which used noise from an analog circuit to generate true random numbers. Other examples include [radioactive decay](#), [thermal noise](#), [shot noise](#) and [clock drift](#). Even [Lava Lamps](#) have been used.

To provide a degree of randomness intermediate between specialized hardware on the one hand and algorithmic generation on the other, some security related computer software requires the user to input a lengthy string of mouse movements, or keyboard input.

Post-processing and statistical checks

Even given a source of plausible random numbers (perhaps from a quantum mechanically based hardware generator), obtaining numbers which are completely unbiased takes care. In addition, behavior of these generators often changes with temperature, power supply voltage, the age of the device, or other outside interference. And a software bug in a pseudo-random number routine, or a hardware bug in the hardware it runs on, may be similarly difficult to detect.

Generated random numbers are sometimes subjected to statistical tests before use to ensure that the underlying source is still working, and then post-processed to improve their statistical properties.

Other considerations

Random numbers uniformly distributed between 0 and 1 can be used to generate random numbers of any desired distribution by passing them through the inverse [cumulative distribution function](#) of the desired distribution. Inverse CDFs are also called [quantile functions](#). To generate a pair of [independent standard normally distributed](#) random numbers (x, y) , one may first generate the [polar coordinates](#) (r, θ) , where $r \sim \chi_2^2$ and $\theta \sim \text{UNIFORM}(0, 2\pi)$ (see [Box-Muller transform](#)).

Some 0 to 1 RNGs include 0 but exclude 1, while others include or exclude both.

The outputs of multiple independent RNGs can be combined (for example, using a bit-wise [XOR](#) operation) to provide a combined RNG at least as good as the best RNG used. [More details about uncorrelated near random bit streams](#).

Computational and hardware random number generators are sometimes combined to reflect the benefits of both kinds. Computational random number generators can typically generate pseudo-random numbers much faster than physical generators can generate true randomness.

Uses of random numbers

Random number generators have several important applications in [gambling](#), [statistical sampling](#), [computer simulation](#), [cryptography](#), etc.

Note that, in general, where unpredictability is paramount--such as in security applications--hardware generators are generally preferred, where feasible, over pseudo-random algorithms.

Low-discrepancy sequences as an alternative

Some computations making use of a random number generator can be summarized as the computation of a total or average value, such as the computation of integrals by the [Monte Carlo method](#). For such problems, it may be possible to find a more accurate solution by the use of so-called [low-discrepancy sequences](#), also called [quasirandom](#) numbers. Such sequences have a definite pattern that fills in gaps evenly, qualitatively speaking; a truly random sequence may, and usually does, leave larger gaps.

Generation from a probability distribution

There are a couple of methods to generate a random number based on a [probability distribution function](#). These methods involve transforming a normal random number in some way. Because of this, these methods work equally well in generating both pseudo-random and true random numbers. One method, called the [inversion method](#), involves integrating up to an area greater than or equal to the random number (which should be generated between 0 and 1 for proper distributions). A second method, called the [acceptance-rejection method](#), involves choosing an x and y value and testing whether the function of x is greater than the y value. If it is, the x value is accepted. Otherwise, the x value is rejected and the algorithm tries again. [3]

Hardware random number generator

In [computing](#), a **hardware random number generator** is an apparatus that generates [random numbers](#) from a physical process. Such devices are often based on microscopic phenomena such as [thermal noise](#) or the [photoelectric effect](#) or other [quantum](#) phenomena. These processes are, in theory, completely unpredictable, and the theory's assertions of unpredictability are subject to experimental test. A quantum-based hardware random number generator typically contains an [amplifier](#) to bring the output of the physical process into the [macroscopic](#) realm, and a [transducer](#) to convert the output into a digital signal.

Random number generators can also be built from macroscopic phenomena, such as [playing cards](#), [dice](#), and the [roulette](#) wheel. The presence of unpredictability in these phenomena can be justified by the theory of unstable [dynamical systems](#) and [chaos theory](#). These theories suggest that even though macroscopic phenomena are deterministic in theory under [Newtonian mechanics](#), real-world systems evolve in ways that cannot be predicted in practice because one would need to know the micro-details of initial conditions and subsequent manipulation or change.

Although dice have been mostly used in [gambling](#), and in recent times as 'randomizing' elements in games (e.g. [role playing games](#)), the [Victorian](#) scientist [Francis Galton](#) described a way to use dice to explicitly generate random numbers for scientific purposes, in 1890. Though some gamblers believe they can control their throws of dice enough to win at [dice games](#) (a claim which has been long debated), no one has produced a way to exploit the claimed effect in either generating or attacking physical randomness sources.

Hardware random number generators are often relatively slow, and they may produce a biased sequence (i.e., some values are more common than others). Whether a hardware random number generator is suitable for a particular application depends on the details of both the application and the generator.

Pseudo-random generators

Most computer "random number generators" are not hardware devices, but are software routines implementing generator [algorithms](#). They are often supplied as [library routines](#) in [programming language](#) implementations, or as part of the [operating system](#). These are more properly called [pseudo-random number generators](#), since, being [finite state machines](#), they cannot produce truly random outputs. Within that limitation, some produce sequences which pass one or several statistical pattern tests for randomness, and may be useful in many circumstances. Several common ones are very poor, not only failing to pass many of the relevant statistical tests, but also being readily, even trivially, predictable.

[Algorithmic information theory](#) defines a sequence of [bits](#) as non-random if it can be produced by some computer program that is shorter than that sequence ([Chaitin-Kolmogorov randomness](#)). Pseudo-random number generators fail that test dramatically. They can usually be programmed in a few thousand bits, but can produce far larger sequences, with periods so long no plausible computer or combination of them could exhaust a single period before the [heat death](#) of the Universe.

There are several informal definitions of [randomness](#), usually based on either a lack of discernible [patterns](#) in a sequence, or the unpredictability of the sequence or various aspects of it by, generally, the most puissant possible adversary. Output from well-designed pseudo-random number generators should pass assorted statistical tests probing for non-randomness (see [NIST Special Publication 800-22](#), [Knuth](#), [The Art of Computer Programming](#), vol. 2, and [RFC 4086](#) for details of many such tests).

The sequences such generators produce always have patterns, in one sense, since the algorithm that generates them has a finite state, and must, eventually, repeat one of those states. Given the original state of the generator, and the implementation of the algorithm, a pseudo-random number generator of this sort is totally predictable. Given even partial knowledge of that state, they are often insecure for many purposes. On the other hand, it has become relatively easy to produce pseudo-random number generators that are guaranteed not to repeat on any conceivable computer within a time-frame that is millions of times longer than the [age of the universe](#). It is an open question whether it is always possible, in some practical way, to distinguish the output of such a pseudo-random number generator from that of a perfectly random source, without

knowledge of the generator's internal state. Much of modern cryptography rests on the assumption that [ciphers](#) can be constructed whose output is indistinguishable from random noise without knowledge of a secret [key](#) used in the algorithm.

Uses

Main article: [Applications of randomness](#)

Unpredictable random numbers were first investigated in the context of [gambling](#), and many randomizing devices such as [dice](#), [shuffling playing cards](#), and [roulette](#) wheels, were first developed for use in gambling. Fairly produced random numbers are vital to electronic gambling and ways of creating them are sometimes regulated by governmental gaming commissions.

Random numbers are also used for non-gambling purposes, both where their use is mathematically important, such as sampling for [opinion polls](#), and in situations where fairness is approximated by [randomization](#), such as selecting [jurors](#) and [military draft lotteries](#). Their use is antique as well. In the [Book of Numbers](#) (33:54), [Moses](#) commands the Israelites to apportion the land by lot (גורל). And the drawing of lots, often pottery shards, are well attested in the Classical world of Greece and Rome. Some of these very shards have been archeologically recovered.

Early attempts

One early way of producing random numbers was by a variation of the same machines used to play [keno](#) or select [lottery](#) numbers. Basically, these mixed numbered ping-pong balls with blown air, perhaps combined with mechanical agitation, and use some method to withdraw balls from the mixing chamber. This method gives reasonable results in some senses, but the random numbers generated by this means are expensive. The method is inherently slow, and is unusable in most automated situations, (i.e., with computers).

In [1927](#), [Cambridge University Press](#) published a book of *Random sampling numbers*, arranged by a statistician, Leonard Henry Caleb Tippett, which contained 41,600 digits taken from English parishes listed in census records. Other random number tables were published in that era, including one by [R. A. Fischer](#) and another by the U.S. [Interstate Commerce Commission](#) in 1949 with over 100,000 random digits.

What is likely to be the last of these projects is [A Million Random Digits with 100,000 Normal Deviates](#), published by the [RAND Corporation](#) in [1955](#). They created an electronic simulation of a roulette wheel attached to a [key punch](#), the results of which were then carefully filtered and tested before being used to generate the table. The RAND table was a significant breakthrough in delivering random numbers because such a large and carefully prepared table had never before been available. It was useful for simulations and modeling. But, having been published, it is not usable as cryptographic keys, or for preparing them, or as a seed in some cryptographic pseudo-random number generator. However, since it was published long before modern cryptography, using values from it for the random (if not unknown) constants for initializing algorithms would

demonstrate that the constants had not been selected for (in [B. Schneier](#)'s words) "nefarious purpose(es)." [Khufu and Khafre](#) do this, for example (ref Applied Cryptography - B. Schneier). See: [Nothing up my sleeve numbers](#).

Physical phenomena with random properties

There are two fundamental sources of practical quantum mechanical physical randomness: quantum mechanics at the atomic or sub-atomic level and [thermal noise](#) (some of which is quantum mechanical in origin). Quantum mechanics predicts that certain physical phenomena, such as the [nuclear decay](#) of atoms, are fundamentally random and cannot, in principle, be predicted. (For a discussion of empirical verification of quantum unpredictability, see [Bell test experiments](#).) And, because we live at a finite, non-zero temperature, every system has some random variation in its state; for instance, molecules of air are constantly bouncing off each other in a random way. (See [statistical mechanics](#).) This randomness is a quantum phenomenon as well. (See [phonon](#).)

Because the outcome of quantum-mechanical events cannot in principle be predicted, they are the 'gold standard' for random number generation. Some quantum phenomena used for random number generation include:

- [Shot noise](#), a quantum mechanical noise source in electronic circuits. A simple example is a lamp shining on a photodiode. Due to the [uncertainty principle](#), arriving photons create noise in the circuit. Collecting the noise for use poses some problems, but this is an especially simple random noise source.
- A [nuclear decay](#) radiation source (as, for instance, from some kinds of commercial [smoke detectors](#)), detected by a [Geiger counter](#) attached to a PC.
- [Photons](#) travelling through a [semi-transparent mirror](#), as in the commercial product, Quantis from id Quantique SA. The [mutually exclusive events](#) (reflection — transmission) are detected and associated to "0" or "1" bit values respectively.

Thermal phenomena are easier to detect. They are (somewhat) vulnerable to attack by lowering the temperature of the system, though most systems will stop operating at temperatures (e.g., ~150 K) low enough to reduce noise by a factor of two. Some of the thermal phenomena used include:

- [thermal noise](#) from a [resistor](#), amplified to provide a random voltage source.
- [Avalanche noise](#) generated from an [avalanche diode](#), or Zener breakdown noise from a reverse-biased [zener diode](#).
- [Atmospheric noise](#), detected by a radio receiver attached to a PC (though much of it, such as lightning noise, is not properly thermal noise, but most likely a [chaotic](#) phenomenon).

Another variable physical phenomenon that is easy to measure is [clock drift](#). It is ultimately related to component differences (due perhaps to quantum effects during manufacture), to design differences (different designs will usually differ in their variance characteristics), to behavior changes caused by component aging, and to configuration / set up differences. Many of these, with sufficient examination of the hardware and its behaviors, can be predicted well enough to render the random numbers produced more or less predictable, and so non-random.

In the absence of quantum effects or thermal noise, other phenomena that tend to be random, although in ways not easily characterized by laws of physics, can be used. When several such sources are combined carefully (as in, for example, the [Yarrow algorithm](#) or [Fortuna CSPRNGs](#)), enough entropy can be collected for the creation of cryptographic keys and [nonces](#), though generally at restricted rates. The advantage is that this approach needs, in principle, no special hardware. The disadvantage is that a sufficiently knowledgeable attacker can surreptitiously modify the software or its inputs, thus reducing the randomness of the output, perhaps substantially. The primary source of randomness typically used in such approaches is the precise timing of the [interrupts](#) caused by mechanical input/output devices, such as keyboards and [disk drives](#), various system information counters, etc.

This last approach must be implemented carefully and may be subject to attack if it is not. For instance, the generator built into the Linux kernel, which combines several such sources, may be vulnerable to an attack. The random number generator used for cryptographic purposes in an early version of the [Netscape](#) browser was certainly vulnerable (and was promptly changed).

One approach in using physical randomness is to convert a noise source into a random bit sequence in a separate device that is then connected to the computer through an I/O port. The acquired noise signal is amplified, filtered, and then run through a high-speed voltage comparator to produce a logic signal that alternates states at random intervals. At least in part, the randomness produced depends on the specific details of the 'separate device'. Care must also always be taken when amplifying low-level noise to keep out spurious signals, such as power line hum and unwanted broadcast transmissions, and to avoid adding bias during acquisition and amplification. In some simple designs, the fluctuating logic value is converted to an [RS-232](#) type signal and presented to a computer's serial port. Software then sees this series of logic values as bursts of "[line noise](#)" characters on an I/O port. More sophisticated systems may format the bit values before passing them into a computer.

Another approach is to feed an analog noise signal to an [analog to digital converter](#), such as the audio input port built into most personal computers. The digitized signal may then be processed further in software to remove bias. However, digitization is itself often a source of bias, sometimes subtle, so this approach requires considerable caution and care.

Some have suggested using digital cameras, such as [webcams](#), to photograph chaotic macroscopic phenomena. A group at [Silicon Graphics](#) imaged [Lava lamps](#) to generate random numbers. [U.S. Patent 5732138](#). One problem was determining whether the chaotic shapes generated were actually random -- the team decided that they are in properly operating Lava lamps. Other chaotic scenes could be employed, such as the motion of streamers in a fan air stream or, probably, bubbles in a [fish tank](#) (fish optional). The digitized image will generally

contain additional noise, perhaps not very random, resulting from the video to digital conversion process. A higher quality device might use two sources and eliminate signals that are common to both— depending on the sources and their physical locations, this reduces or eliminates interference from outside electric and magnetic fields. This is often recommended for gambling devices, to reduce cheating by requiring attackers to exploit bias in several "random bit" streams.

The [Intel](#) 80802 Firmware Hub chip included a hardware RNG using two free running oscillators, one fast and one slow. A thermal noise source (non-commonmode noise from two diodes) is used to modulate the frequency of the slow oscillator, which then triggers a measurement of the fast oscillator. That output is then debiased using a [von Neumann](#) type decorrelation step (see below). The output rate of this device is somewhat less than 100,000 bps. This chip was an optional component of the 840 chipset family that supported an earlier Intel bus. It is not included in modern PCs.

All [VIA C3](#) microprocessors have included a hardware RNG on the processor chip since 2003. Instead of using thermal noise, raw bits are generated by using four freerunning oscillators which are designed to run at different rates. The output of two are XORed to control the bias on a third oscillator, whose output clocks the output of the fourth oscillator to produce the raw bit. Minor variations in temperature, silicon characteristics, and local electrical conditions cause continuing oscillator speed variations and thus produce the entropy of the raw bits. To further insure randomness, there are actually two such RNGs on each chip, each positioned in different environments and rotated on the silicon. The final output is a mix of these two generators. The raw output rate is tens to hundreds of megabits per second, and the whitened rate is a few megabits per second. User software can access the generated random bit stream using new non-privileged machine language instructions.

A software implementation of a related idea on ordinary hardware is included in *CryptoLib*, a cryptographic routine library (JB Lacy, DP Mitchell, WM Schell, *CryptoLib: Cryptography in software*, Proc 4th USENIX Security Symp, pg 1-17, 1993). The algorithm is [truerand](#). Most modern computers have two crystal oscillators, one for the real-time clock and one for the primary CPU clock; *truerand* exploits this fact. It uses an operating system service that sets an alarm, running off the real-time clock. One subroutine sets that alarm to go off in one clock tick (usually 1/60th of a second). Another then enters a while loop waiting for the alarm to trigger. Since the alarm will not always trigger in exactly one tick, the least significant bit of a count of loop iterations, between setting the alarm and its trigger, will vary randomly, possibly enough for some uses. *truerand* doesn't require additional hardware, but in a multi-tasking system great care must be taken to avoid non-randomizing interference from other processes (e.g., in the suspension of the counting loop process as the operating system scheduler starts and stops assorted processes).

Dealing with bias

The bit-stream from such systems is prone to be biased, with either 1s or 0s predominating. There are two approaches to dealing with bias and other artifacts. The first is to design the RNG to minimize bias inherent in the operation of the generator. One method to correct this feeds back the generated bit stream, filtered by a low-pass filter, to adjust the bias of the generator. By the

[central limit theorem](#), the feedback loop will tend to be well-adjusted 'almost all the time'. Ultra-high speed random number generators often use this method. Even then, the numbers generated are usually somewhat biased.

Software whitening

A second approach to coping with bias is to reduce it after generation (in software or hardware). Even if the above hardware bias reduction steps have been taken, the bit-stream should still be assumed to contain bias and correlation. There are several techniques for reducing bias and correlation, often known by the name "[whitening](#)" algorithms, by analogy with the related problem of producing white noise from a correlated signal.

[John von Neumann](#) invented a simple algorithm to fix simple bias, and reduce correlation. It considers bits two at a time, taking one of three actions: when two successive bits are the same, they are not used as a random bit, a sequence of 1,0 becomes a 1, and a sequence of 0,1 becomes a zero. This eliminates simple bias, and is easy to implement as a computer program or in digital logic. This technique works no matter how the bits have been generated. It cannot assure randomness in its output, however. What it can do (with significant numbers of discarded bits) is transform a random bit stream with a frequency of 1's different from 50% into a stream closer to that frequency.

Another technique for improving a near random bit stream is to [exclusive-or](#) the bit stream with the output of a high-quality [cryptographically secure pseudorandom number generator](#) such as [Blum Blum Shub](#) or a good [stream cipher](#). This can cheaply improve decorrelation and digit bias.

A related method which reduces bias in a near random bit stream is to take two or more uncorrelated near random bit streams, and [exclusive or](#) them together. Let the probability of a bit stream producing a 0 be $1/2 + e$, where $-1/2 \leq e \leq 1/2$. Then e is the bias of the bitstream. If two uncorrelated bit streams with bias e are exclusive-or-ed together, then the bias of the result will be $2e^2$. This may be repeated with more bit streams. (See also [Piling-up lemma](#)).

A very simple (and fast) technique that is only viable when an equal number of 1 and 0 is desired is to flip every other bit from the output stream of the RNG. Even if there is a strong bias in the hardware RNG, such as 85%, statistically half of the 0s will be turned into 1s and vice-versa, leaving a perfectly-biased system (statistically speaking). This technique works if the input stream is or is not biased, but only if the stream is truly random.

Some designs apply cryptographic [hash functions](#) such as [MD5](#), [SHA-1](#), or [RIPEMD-160](#) or even a [CRC](#) function to all or part of the bit stream, and then use the output as the random bit stream. This is attractive, partly because it is relatively fast compared to some other methods, but depends entirely on qualities in the hash output for which there may be little theoretical basis.

PRNG with periodically refreshed random key

Other designs use what are believed to be true random bits as the [key](#) for a high quality [block cipher](#) algorithm, taking the encrypted output as the random bit stream. Care must be taken in

these cases to select an appropriate [block mode](#), however. In some implementations, the PRNG is run for a limited number of digits, while the hardware generating device produces a new seed.

Using observed events

Software engineers without true random number generators often try to develop them by measuring physical events available to the software. An example is measuring the time between user keystrokes, and then taking the least significant bit (or two or three) of the count as a random digit. A similar approach measures task-scheduling, network hits, disk-head seek times and other internal events. One Microsoft design includes a very long list of such internal values (see the [CSPRNG](#) article).

The method is risky when it uses computer-controlled events because a clever, malicious attacker might be able to predict a cryptographic key by controlling the external events. Several gambling frauds have been uncovered which rely on manipulating normally hidden events internal to the operation of computers or networks. It is also risky because the supposed user-generated event (e.g., keystrokes) can be [spoofed](#) by a sufficiently ingenious attacker, allowing control of the "random values" used by the cryptography.

However, with sufficient care, a system can be designed that produces cryptographically secure random numbers from the sources of randomness available in a modern computer. The basic design is to maintain an "entropy pool" of random bits that are assumed to be unknown to an attacker. New randomness is added whenever available (for example, when the user hits a key) and an estimate of the number of bits in the pool that cannot be known to an attacker is kept. Some of the strategies in use include:

- When random bits are requested, return that many bits derived from the entropy pool (by a cryptographic hash function, say) and decrement the estimate of the number of random bits remaining in the pool. If not enough unknown bits are available, wait until enough are available. This is the top-level design of the ["/dev/random"](#) device in Linux, written by [Theodore Ts'o](#) and used in many other Unix-like operating systems. It provides high-quality random numbers so long as the estimates of the input randomness are sufficiently cautious. The Linux ["/dev/urandom"](#) device is a simple modification which disregards estimates of input randomness, and is therefore rather less likely to have high entropy as a result.
- Maintain a [stream cipher](#) with a key and [IV](#) obtained from an entropy pool. When enough bits of entropy have been collected, replace both key and IV with new random values and decrease the estimated entropy remaining in the pool. This is the approach taken by the [yarrow](#) library. It provides resistance against some attacks and conserves hard-to-obtain entropy.

Problems

It is very easy to misconstruct hardware or software devices which attempt to generate random numbers. Also, most 'break' silently, often producing decreasingly random numbers as they

degrade. A physical example might be the rapidly decreasing radioactivity of the smoke detectors mentioned earlier. Failure modes in such devices are plentiful and are neither easy, quick, nor inexpensive to detect.

Because many entropy sources are often quite fragile, and fail silently, statistical tests on their output should be performed continuously. Many, but not all, such devices include some such tests into the software that reads the device.

Just as with other components of a cryptosystem, a software random number generator should be designed to resist certain attacks. Defending against these attacks is difficult.

Estimating entropy

There are mathematical techniques for estimating the [entropy](#) of a sequence of symbols. None are so reliable that their estimates can be fully relied upon; there are always assumptions which may be very difficult to confirm. These are useful for determining if there is enough entropy in a seed pool, for example, but they cannot, in general, distinguish between a true random source and a pseudo-random generator.

Performance checks

Hardware random number generators should be constantly monitored for proper operation. [RFC 4086](#) and [FIPS Pub 140-2](#) include tests which can be used for this. Also see the documentation for the New Zealand cryptographic software library [cryptlib](#).

Since many practical designs rely on a hardware source as an input, it will be useful to at least check that the source is still operating. Statistical tests can often detect failure of a noise source, such as a radio station transmitting on a channel thought to be empty, for example. Noise generator output should be sampled for testing before being passed through a "whitener." Some whitener designs can pass statistical tests with no random input. While detecting a large deviation from perfection would be a sign that a true random noise source has become degraded, small deviations are normal and can be an indication of proper operation. Correlation of bias in the inputs to a generator design with other parameters (e.g., internal temperature, bus voltage) might be additionally useful as a further check. Unfortunately, with currently available (and foreseen) tests, passing such tests is not enough to be sure the output sequences are random. A carefully chosen design, verification that the manufactured device implements that design and continuous physical security to insure against tampering may all be needed in addition to testing for high value uses.