Cryptographically secure pseudorandom number generator

A **cryptographically secure pseudo-random number generator** (**CSPRNG**) is a pseudo-random number generator (PRNG) with properties that make it suitable for use in cryptography.

Many aspects of cryptography require random numbers, for example:

- Key generation
- Nonces
- One-time pads
- Salts in certain signature schemes, including ECDSA, RSASSA-PSS.

The "quality" of the randomness required for these applications varies. For example creating a nonce in some protocols needs only uniqueness. On the other hand, generation of a master key requires a higher quality, such as more entropy. And in the case of one-time pads, the information-theoretic guarantee of perfect secrecy only holds if the key material is obtained from a true random source with high entropy.

Ideally, the generation of random numbers in CSPRNGs uses entropy obtained from a high quality source, which might be a hardware random number generator or perhaps unpredictable system processes — though unexpected correlations have been found in several such ostensibly independent processes. From an information theoretic point of view, the amount of randomness, the entropy that can be generated is equal to the entropy provided by the system. But sometimes, in practical situations, more random numbers are needed than there is entropy available. Also the processes to extract randomness from a running system are slow in actual practice. In such instances, a CSPRNG can sometimes be used. A CSPRNG can "stretch" the available entropy over more bits.

When all the entropy we have is available before algorithm execution begins, we really have a stream cipher. However some crypto system designs allow for the addition of entropy during execution, in which case it is not a stream cipher equivalent and cannot be used as one. Stream cipher and CSPRNG design is thus closely related.

Requirements

The requirements of an ordinary PRNG are also satisfied by a cryptographically secure PRNG, but the reverse is not true. CSPRNG requirements fall into two groups: first, that they pass statistical randomness tests; and secondly, that they hold up well under serious attack, even when part of their initial or running state becomes available to an attacker.

- Every CSPRNG should satisfy the "next-bit test". The next-bit test is as follows: Given the first *k* bits of a random sequence, there is no polynomial-time algorithm that can predict the (*k*+1)th bit with probability of success better than 50%. Andrew Yao proved in 1982 that a generator passing the next-bit test will pass all other polynomial-time statistical tests for randomness.

- Every CSPRNG should withstand "state compromise extensions". In the event that part or all of its state has been revealed (or guessed correctly), it should be impossible to reconstruct the stream of random numbers prior to the revelation. Additionally, if there is an entropy input while running, it should be infeasible to use knowledge of the input's state to predict future conditions of the CSPRNG state.

  Example: If the CSPRNG under consideration produces output by computing bits of π in sequence, starting from some unknown point in the binary expansion, it may well satisfy the next-bit test and thus be statistically random, as π appears to be a random sequence. (This would be guaranteed if π is a normal number, for example.) However, this algorithm is not cryptographically secure; an attacker who determines which bit of pi (i.e. the state of the algorithm) is currently in use will be able to calculate all preceding bits as well.

Most PRNGs are not suitable for use as CSPRNGs and will fail on both counts. First, while most PRNGs outputs appear random to assorted statistical tests, they do not resist determined reverse engineering. Specialized statistical tests may be found specially tuned to such a PRNG that shows the random numbers not to be truly random. Second, for most PRNGs, when their state has been revealed, all past random numbers can be retrodicted, allowing an attacker to read all past messages, as well as future ones.

CSPRNGs are designed explicitly to resist this type of cryptanalysis.

Some background

Santha and Vazirani proved that several bit streams with weak randomness can be combined to produce a higher-quality quasi-random bit stream. Even earlier, John von Neumann proved that a simple algorithm can remove a considerable amount of the bias in any bit stream which should be applied to each bit stream before using any variation of the Santha-Vazirani design. The field is termed *entropy extraction* and is the subject of active research (e.g., N Nisan, S Safra, R Shaltiel, A Ta-Shma, C Umans, D Zuckerman).

Designs

In the discussion below, CSPRNG designs are divided into three classes: 1) those based on cryptographic primitives such as ciphers and cryptographic hashes, 2) those based upon mathematical problems thought to be hard, and 3) special-purpose designs. The last often introduce additional entropy when available and, strictly speaking, are not "pure" pseudorandom number generators, as their output is not completely determined by their initial state. This addition can prevent attacks even if the initial state is compromised.

Designs based on cryptographic primitives

- A secure block cipher can be converted into a CSPRNG by running it in counter mode. This is done by choosing a random key and encrypting a zero, then encrypting a 1, then encrypting a 2, etc. The counter can also be started at an arbitrary number other than zero. Obviously, the period will be 2^n for an n-bit block cipher; equally obviously, the initial values (i.e., key and "plaintext") must not become known to an attacker,however good this CSPRNG construction might be. Otherwise, all security will be lost.

- A cryptographically secure hash of a counter might also act as a good CSPRNG in some cases. In this case, it is also necessary that the initial value of this counter is random and secret. However, there has been little study of these algorithms for use in this manner, and at least some authors warn against this use.[3]

- Most stream ciphers work by generating a pseudorandom stream of bits that are combined (almost always XORed) with the plaintext; running the cipher on a counter will return a new pseudorandom stream, possibly with a longer period. The cipher is only secure if the original stream is a good CSPRNG (this is not always the case: see RC4 cipher). Again, the initial state must be kept secret.

Number theoretic designs

- The Blum Blum Shub algorithm has a security proof, based on the difficulty of the Quadratic residuosity problem. Since the only known way to solve that problem is to factor the modulus, it is generally regarded that the difficulty of integer factorization provides a conditional security proof for the Blum Blum Shub algorithm. However the algorithm is very inefficient and therefore impractical unless really extreme security is needed.
- The Blum-Micali algorithm has an unconditional security proof based on the difficulty of the discrete logarithm problem but is also very inefficient.

Special designs

There are a number of practical PRNGs that have been designed to be cryptographically secure, including

- the Yarrow algorithm which attempts to evaluate the entropic quality of its inputs. Yarrow is used in FreeBSD, OpenBSD and Mac OS X (also as /dev/random)
- the Fortuna algorithm, the successor to Yarrow, which does not attempt to evaluate the entropic quality of its inputs.
- the function CryptGenRandom provided in Microsoft's Cryptographic Application Programming Interface
- ISAAC based on a variant of the RC4 cipher
- ANSI X9.17 standard (*Financial Institution Key Management (wholesale)*), which has been adopted as a FIPS standard as well. It takes as input a TDEA (keying option 2) key

bundle k and (the initial value of) a 64 bit random seed s. Each time a random number is required it:

- Obtains the current date/time D to the maximum resolution possible.
- Computes a temporary value $t = \mathrm{TDEA}_k(D)$
- Computes the random value $x = \mathrm{TDEA}_k(s \oplus t)$, where $\oplus$ denotes bitwise exclusive or.
- Updates the seed $s = \mathrm{TDEA}_k(x \oplus t)$

Obviously, the technique is easily generalized to any block cipher; AES has been suggested (Young and Yung, op cit, sect 3.5.1).

Standards

Several CSPRNGs have been standardized. For example,

- FIPS 186-2
- NIST SP 800-90: Hash_DRBG, HMAC_DRBG, CTR_DRBG and Dual EC DRBG.
- ANSI X9.17-1985 Appendix C
- ANSI X9.31-1998 Appendix A.2.4
- ANSI X9.62-1998 Annex A.4, obsoleted by ANSI X9.62-2005, Annex D (HMAC_DRBG)

A good reference is maintained by NIST.

There are also standards for statistical testing of new CSPRNG designs:

- *A Statistical Test Suite for Random and Pseudorandom Number Generators*, NIST Special Publication 800-22.

Pseudorandom number generator

A **pseudorandom number generator** (**PRNG**), also known as a **deterministic random bit generator** (**DRBG**), is an algorithm for generating a sequence of numbers that approximates the properties of random numbers. The sequence is not truly random in that it is completely determined by a relatively small set of initial values, called the PRNG's *state*, which includes a truly random seed. Although sequences that are closer to truly random can be generated using hardware random number generators, *pseudorandom* numbers are important in practice for their speed in number generation and their reproducibility, and they are thus central in applications such as simulations (e.g., of physical systems with the Monte Carlo method), in cryptography, and in procedural generation. Good statistical properties are a central requirement for the output of a PRNG, and common classes of suitable algorithms include linear congruential generators, lagged Fibonacci generators, and linear feedback shift registers. Cryptographic applications require the output to also be unpredictable, and more elaborate designs, which do not inherit the linearity of simpler solutions, are needed. More recent instances of PRNGs with strong randomness guarantees are based on computational hardness assumptions, and include the Blum Blum Shub, Fortuna, and Mersenne Twister algorithms.

In general, careful mathematical analysis is required to have any confidence that a PRNG generates numbers that are sufficiently "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."Robert R. Coveyou of Oak Ridge National Laboratory once titled an article, "The generation of random numbers is too important to be left to chance."

Periodicity

A PRNG can be started from an arbitrary starting state using a seed state. It will always produce the same sequence thereafter when initialized with that state. The maximum length of the sequence before it begins to repeat is determined by the size of the state, measured in bits. However, since the length of the maximum 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. Mixes (no restrictions) have periods of about $2^{n/2}$ on average, usually after walking through a nonrepeating starting sequence. Mixes that are reversible (permutations) have periods of about 2^{n-1} on average, and the period will always include the original internal state. 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 1-bit output.

Most pseudorandom generator 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.

Problems with deterministic generators

In practice, the output from many common PRNGs exhibit artifacts which 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 amounts 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,[*clarification needed*] but its inadequacy went undetected for a very long time. In many fields, much research work of that period which relied on random selection or on Monte Carlo style simulations, or in other ways, is less reliable than it might have been as a result.

Early approaches

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 available then, 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.

Mersenne Twister

The 1997 invention of the Mersenne Twister algorithm, by Makoto Matsumoto and Takuji Nishimura, avoids many of the problems with earlier generators. It has the colossal period of

$2^{19937}-1$ iterations ($>4.3\times10^{6,001}$), is proven to be equidistributed in (up to) 623 dimensions (for 32-bit values), and runs faster than other statistically reasonable generators. It is now increasingly becoming the random number generator of choice for statistical simulations and generative modeling. SFMT, SIMD-oriented Fast Mersenne Twister, a variant of Mersenne Twister, is faster even if it's not compiled with SIMD support.

The native Mersenne Twister is not considered suitable for use in all cryptographic applications. A variant of Mersenne Twister has been proposed as a cryptographic cipher.

Cryptographically secure pseudorandom number generators

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. 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 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.

BSI evaluation criteria

The German Federal Office for Information Security (BSI) has established four criteria for quality of deterministic random number generators. 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 BSI2 (AIS 20, v. 1, 1999) and FIPS (140-1, 1994), 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.