Random Number Generator Recommendations for Applications

This version of the document is dated 2021-11-14.

Peter Occil

Most apps that use randomly generated or pseudorandom numbers care about either unpredictability, high quality, or repeatability. This article gives recommendations on choosing the right kind of random number generator (RNG) or pseudorandom number generator (PRNG) for the application.

1 Introduction

Many applications rely on random number generators (RNGs) to produce a sequence of numbers that seemingly occur by chance; however, it's not enough for this sequence to merely "look random". But unfortunately, most popular programming languages today—

- specify few and weak requirements on their built-in RNGs (such as **C's `rand`**),
- specify a relatively weak general-purpose RNG (such as Java's `java.math.Random`),
- implement RNGs by default that leave something to be desired (such as Mersenne Twister),
- initialize RNGs with a timestamp by default (such as the **.NET Framework implementation of `System.Random`**), and/or
- use RNGs that are initialized with a fixed value by default (as is the case in **MATLAB** and C[^1]),

so that as a result, many applications use RNGs, especially built-in RNGs, that have little assurance of high quality or security. That is why this document discusses high-quality RNGs and suggests **existing implementations** of them.

This document covers:

- Cryptographic RNGs[^2], noncryptographic RNGs, and manually-seeded pseudorandom number generators, as well as recommendations on their use and properties.
- Nondeterministic sources, entropy, and seed generation.
- Existing implementations of RNGs.
- Guidance for implementations of RNGs designed for reuse by applications.
- Issues on shuffling with an RNG.

This document does not cover:

- Testing an RNG implementation for correctness[^3] or statistical quality. See my document on **testing PRNGs**.
- Generating numbers with unequal probabilities; I discuss this topic in **another document**.
- Generators of low-discrepancy sequences (quasirandom sequences), such as Sobol sequences. They are not RNGs since the numbers they produce depend on prior results.
- Applications for which the selection of RNGs is limited by regulatory requirements.

1.1 About This Document

This is an open-source document; for an updated version, see the source code or its rendering on GitHub. You can send comments on this document either on CodeProject or on the GitHub issues page.

2 Contents

3 Definitions

In this document:

- **Random number generator (RNG)** means software and/or hardware that seeks to generate integers in a bounded range such that each possible outcome is as likely as

any other without influence by anything else[^4].
- **Pseudorandom number generator (PRNG)** means a random number generator that produces numbers by an algorithm that mathematically expands its input.
- **Seed** means arbitrary data serving as a PRNG's input.
- **Information security** means keeping information safe from attacks that could access, use, delay, or manipulate that information.[^5]

4 Summary

- Does the application use random-behaving numbers for **information security** purposes (e.g., as passwords or other secrets)?
  - Yes: Use a **cryptographic RNG**.
- No: Does the application require **reproducible "random" numbers**?
  - Yes: Use a manually-seeded high-quality PRNG. If a seed is known, use it. Otherwise, generate a fresh seed using a cryptographic RNG.
    - Does the application run **multiple independent processes** that use pseudorandom numbers?
      - No: Seed one PRNG with the seed determined above.
      - Yes: Pass the seed determined above to each process as described in "**Seed Generation for Noncryptographic PRNGs**".
- No: Is a cryptographic RNG **too slow** for the application?
  - Yes: Use a **high-quality PRNG** with a seed generated using a cryptographic RNG.
  - No: Use a cryptographic RNG.

5 Cryptographic RNGs

Cryptographic RNGs (also known as "cryptographically strong" or "cryptographically secure" RNGs) seek to generate numbers that not only "look random", but are cost-prohibitive to guess. An application should use a cryptographic RNG whenever the application—

- generates random-behaving numbers for information security purposes, or
- generates random-behaving numbers so infrequently that the RNG's speed is not a concern.

See "**Cryptographic RNGs: Requirements**" for requirements.
See "**Existing RNG APIs in Programming Languages**" for existing APIs.
For cryptographic RNGs, an application should use only one thread-safe instance of the RNG for the entire application to use.

Examples: A cryptographic RNG is recommended—

- when generating security parameters (including encryption keys, random passwords, nonces, session identifiers, "salts", and secret values),
- for the purposes of sending or receiving messages or other data securely between computers, or
- whenever predicting future random outcomes would give a player or user a significant and unfair advantage (such as in multiplayer networked games).

6 Noncryptographic PRNGs

Noncryptographic PRNGs vary widely in the quality of randomness of the numbers they generate. For this reason, a noncryptographic PRNG should not be used—

- for information security purposes (e.g., to generate random passwords, encryption keys, or other secrets),
- if cryptographic RNGs are fast enough for the application, or
- if the PRNG is not *high quality* (see "**High-Quality RNGs: Requirements**").

Noncryptographic PRNGs can be *automatically seeded* (a new seed is generated upon PRNG creation) or *manually seeded* (the PRNG uses a predetermined seed).

- See "**When to Use a Manually-Seeded PRNG**" to learn which kind of seeding to use.
- See "**Seed Generation for Noncryptographic PRNGs**" for advice on how to seed.
- See "**Existing RNG APIs in Programming Languages**" for existing APIs.
- For automatically-seeded PRNGs, an application should use only one instance of the generator and pass it around to parts of the application that need it.

7 Manually-Seeded PRNGs

A given pseudorandom number generator (PRNG) generates the same sequence of "random" numbers for the same "seed". Some applications care about reproducible "randomness" and thus could set a PRNG's seed manually for reproducible "random" numbers.

7.1 When to Use a Manually-Seeded PRNG

By seeding a PRNG manually for reproducible "randomness", an application will be tied to that PRNG or its implementation. For this reason, an application should not use a manually-seeded PRNG (rather than a cryptographic or automatically-seeded RNG) unless —

1. the application might need to generate the same "random" result multiple times,
2. the application either—
    - makes the seed (or a "code" or "password" based on the seed) accessible to the user, or
    - finds it impractical to store or distribute the "random" numbers or "random" content, rather than the seed, for later use (e.g., to store those numbers to "replay" later, to store that content in a "save file", or to distribute that content rather than a seed to networked users), and
3. any feature that uses such a PRNG to generate that "random" result is reproducible, in that it produces the same "random" result for the same seed for as long as the feature is still in use by the application.

7.2 Manually-Seeded PRNG Recommendations

If an application chooses to use a manually-seeded PRNG for reproducible "randomness", the application—

- should choose a **high-quality PRNG**,
- should choose a PRNG implementation with consistent behavior that will not change in the future,
- ought to document the chosen PRNG being used as well as all the parameters for that PRNG, and

- should not seed the PRNG with floating-point numbers or generate floating-point numbers with that PRNG.

For advice on generating seeds for the PRNG, see "**Seed Generation for Noncryptographic PRNGs**").

> **Example:** An application could implement a manually-seeded PRNG using a third-party library that specifically says it implements a **high-quality PRNG algorithm**, and could initialize that PRNG using a bit sequence from a cryptographic RNG. The developers could also mention the use of the specific PRNG chosen on any code that uses it, to alert other developers that the PRNG needs to remain unchanged.

7.3 Manually-Seeded PRNG Use Cases

Use cases for manually-seeded PRNGs include the following:

- Simulations and machine learning. This includes physics simulations and artificial intelligence (AI) in games, as well as simulations to reproduce published research data.
- Monte Carlo estimations.
- Procedural noise generation.
- Games that generate "random" content that is impractical to store.
- Unit tests in which "randomness" ought not to influence whether they pass or fail. Here, a manually-seeded PRNG with a fixed seed is used in place of another kind of RNG for the purpose of the test, to help ensure consistent results across the computers under test.

7.4 Manually-Seeded PRNGs in Games

Many kinds of game software generate seemingly "random" game content that might need to be repeatedly regenerated, such as—

- procedurally generated maps for a role-playing game,
- **shuffling** a virtual deck of cards for a solitaire game, or
- a randomly chosen configuration of a game board or puzzle board.

In general, the bigger that "random" content is, the greater the justification to use a manually-seeded PRNG and a custom seed to generate that content. The following are special cases:

1. If the game needs reproducible "random" content only at the start of the game session (e.g., a "random" game board or a "random" order of virtual cards) and that content is small (say, no more than a hundred numbers):
    - The game should not use a manually-seeded PRNG unless the seed is based on a "code" or "password" entered by the user. This is a good sign that the game ought to store the "random" content instead of a seed.
2. In a networked game where multiple computers (e.g., multiple players, or a client and server) have a shared view of the game state and numbers from an RNG or PRNG are used to update that game state:
    - The game should not use a manually-seeded PRNG where predicting a random outcome could give a player a significant and unfair advantage (e.g., the random outcome is the result of a die roll, or the top card of the draw pile, for a board or card game). The game may use such a PRNG in other cases to ensure the game state is consistent among computers, including in physics simulations and AI.

Examples:

1. Suppose a game generates a map with random terrain (which uses an RNG to produce lots of numbers) and shows the player a "code" to generate that map (such as a barcode or a string of letters and digits). In this case, the game—
    - may change the algorithm it uses to generate random maps, but
    - should use, in connection with the new algorithm, "codes" that can't be confused with "codes" it used for previous algorithms, and
    - should continue to generate the same random map using an old "code" when the player enters it, even after the change to a new algorithm.
2. Suppose a game implements a chapter that involves navigating a randomly generated dungeon with randomly scattered monsters and items. If the layout of the dungeon, monsters, and items has to be the same for a given week and for all players, the game can seed a PRNG with a hash code generated from the current week, the current month, the current year, and, optionally, a constant sequence of bits.

7.5 Single Random Value

If an application requires only one random value, with a fixed number of bits, then the application can pass the seed to a hash function rather than a PRNG. Examples of this include the following:

- Generating a color pseudorandomly, by passing the seed to the MD5 hash function, which outputs a 128-bit hash code, and taking the first 24 bits of the hash code as the random color.
- Generating a pseudorandom number in a GLSL (OpenGL Shading Language) fragment shader by passing the fragment coordinates (which vary for each fragment, or "pixel") as well as a seed (which is the same for all fragments) to the Wang hash, which outputs a 32-bit integer.[^6]

7.6 Ensuring Reproducibility

To ensure that a manually-seeded PRNG delivers reproducible "random" numbers across computers, across runs, and across application versions, an application needs to take special care. Reproducibility is often not achievable if the application relies on features or behavior outside the application's control, including any of the following:

- **Floating-point numbers** are a major source of varying results. Different implementations of the same floating-point operation might have subtle differences even if they're given the same input.[^7] It is nontrivial to control for all of these differences, and they include:
    - Differences in accuracy, as with Java's `Math` vs. `StrictMath`, or the x87 `FSIN` instruction vs. a software implementation of sine.
    - Differences in rounding. Results can vary if the application can't control how floating-point numbers are rounded.[^8]
    - Differences in operation order. Unlike with integers or fixed-point numbers[^9], adding or multiplying floating-point numbers in a different order can change the result. This can happen, for example, with parallel reductions (such as parallel sums and dot products), which split a calculation across several parallel tasks and combine their results in the end. Results can vary, even across runs, if a program automatically chooses whether and how to use parallel reduction.
- **Multithreading** and dynamic task scheduling can cause pseudorandom numbers to

be generated in a different order or by different threads from one run to the next, causing inconsistent results; this can happen even if each thread by itself produces the same pseudorandom numbers for the same input (Leierson et al., 2012)[^10]. Dealing with this issue requires either using a single thread, or assigning PRNGs to individual tasks rather than threads or the whole application.
- **Nondeterministic sources** (where the output can vary even if input and state are the same), such as the file system or the system clock.
- **Undocumented, undefined, or implementation-dependent behavior** or features, including a particular hash table traversal order or a particular size for C/C++'s `int` or `long`.

Thus, an application ought to use manually-seeded PRNGs only when necessary, to minimize the need for reproducible "randomness". Where reproducibility is required, the application ought to avoid floating-point numbers, nondeterministic features, and other behavior outside its control, and ought to stick to the same versions of algorithms it uses.

As for reproducible PRNGs, `java.util.Random` is one example of a PRNG with consistent behavior, but none of the following is such a PRNG:

- The C `rand` **method**, as well as C++'s distribution classes from `<random>`, such as `std::uniform_int_distribution`, use implementation-defined algorithms for pseudorandom number generation.
- .NET's `System.Random` has pseudorandom number generation behavior that could change in the future.

8 Nondeterministic Sources and Seed Generation

RNGs ultimately rely on so-called *nondeterministic sources*; without such sources, no computer can produce numbers at random.

8.1 What Is a Nondeterministic Source?

A *nondeterministic source* is a source that doesn't give the same output for the same input each time (for example, a clock that doesn't always give the same time). There are many kinds of them, but sources useful for generating numbers at random have hard-to-guess output (that is, they have high *entropy*; see the next section). They include—

- timings of interrupts and disk accesses,
- timings of keystrokes and/or other input device interactions,
- thermal noise,
- the output generated with A. Seznec's technique called hardware volatile entropy gathering and expansion (HAVEGE), provided a high-resolution counter is available, and
- differences between two high-resolution counter values taken in quick succession (such as in "Jitter RNG"; see (Müller)[^11]).

RFC 4086, "Randomness Requirements for Security", section 3, contains a survey of nondeterministic sources.

> **Note:** Online services that make randomly generated numbers available to applications, as well as the noise registered by microphone and camera recordings (see RFC 4086 sec. 3.2.1, (Liebow-Feeser 2017a)[^12], and (Liebow-Feeser 2017b)[^13]), are additional nondeterministic sources. However, online

services require Internet or other network access, and some of them require access credentials. Also, many mobile operating systems require applications to declare network, camera, and microphone access to users upon installation. For these reasons, these kinds of sources are not recommended if other approaches are adequate.

Example: A program could ask users to flip coins or roll dice and type in their results. If users do so, the results typed this way will have come from nondeterministic sources (here, coins or dice).

8.2 What Is Entropy?

Entropy is a value that describes how hard it is to guess a nondeterministic source's output, compared to an ideal process of generating independent uniform random bits. Entropy is generally the number of bits produced by that ideal process. (For example, a 64-bit output with 32 bits of entropy is as hard to guess as 32 independent uniform random bits.) NIST SP 800-90B recommends *min-entropy* as the entropy measure. Characterizing a nondeterministic source's entropy is nontrivial and beyond the scope of this document. See also RFC 4086 section 2.

8.3 Seed Generation

In general, there are two steps to generate an `N`-bit seed for a PRNG[^14]:

1. Gather enough data from independent nondeterministic sources to reach `N` bits of *entropy* or more.
2. Then, condense the data into an `N`-bit number, a process called *randomness extraction*.

See my **Note on Randomness Extraction**. It should be mentioned, though, that in information security applications, unkeyed hash functions should not be used by themselves in randomness extraction.

8.4 Seed Generation for Noncryptographic PRNGs

In general, to generate a seed allowed by a noncryptographic PRNG, an application ought to use a cryptographic RNG or a method described in the **previous section**.

It is not recommended to seed PRNGs with timestamps, since they can carry the risk of generating the same "random" number sequence accidentally.[^15]

8.4.1 Seeding Multiple Processes

Some applications require multiple processes (including threads, tasks, or subtasks) to use **reproducible "random" numbers** for the same purpose. An example is multiple instances of a simulation with random starting conditions. However, noncryptographic PRNGs tend to produce number sequences that are correlated to each other, which is undesirable for simulations in particular.

To reduce this correlation risk, the application can choose a **high-quality PRNG that supports *streams*** of uncorrelated sequences (nonoverlapping sequences that behave like sequences of numbers chosen uniformly and independently at random) and has an efficient way to assign a different *stream* to each process. For example, in some PRNGs, these *streams* can be formed—

- by initializing PRNGs with consecutive seeds (as in "counter-based" PRNGs (Salmon et al. 2011)[^16]), or
- by discarding a fixed but huge number of PRNG outputs in an efficient way ("**jump-ahead**").

Multiple processes can be seeded for pseudorandom number generation as follows.[^17]

1. **Stream case.** If the PRNG supports *streams* as described above: Generate a seed (or use a predetermined seed), then:

    1. Create a PRNG instance for each process.
    2. Hash the seed and a fixed identifier to generate a new seed allowed by the PRNG.
    3. For each process, advance the PRNG to the next stream (unless it's the first process), then give that process a copy of the PRNG's current internal state.

2. **General case.** For other PRNGs, or if each process uses a different PRNG design, the following is a way to seed multiple processes for pseudorandom number generation, but it carries the risk of generating seeds that lead to overlapping, correlated, or even identical number sequences, especially if the processes use the same PRNG.[^18] Generate a seed (or use a predetermined seed), then:

    1. Create a PRNG instance for each process. The instances need not all use the same PRNG design or the same parameters; for example, some can be SFC64 and others `xoroshiro128**`.
    2. For each process, hash the seed, a unique number for that process, and a fixed identifier to generate a new seed allowed by the process's PRNG, and initialize that PRNG with the new seed.

3. **Leapfrogging** (Bauke and Mertens 2007)[^19]. The following is an alternative way to initialize a PRNG for each process if the number of processes (`N`) is small. Generate a seed (or use a predetermined seed), then:

    1. Create one PRNG instance. Hash the seed and a fixed identifier to generate a new seed allowed by the PRNG.
    2. Give each process a copy of the PRNG's state. Then, for the *second* process, discard 1 output from its PRNG; for the *third* process, discard 2 outputs from its PRNG; and so on.
    3. Now, whenever a PRNG created this way produces an output, it then discards the next `N` minus 1 outputs before finishing.

    Note: The steps above include hashing several things to generate a new seed. This has to be done with either a **hash function** of N or more bits (where N is the PRNG's maximum seed size), or a so-called "seed sequence generator" like C++'s `std::seed_seq`.[^20]

    Examples:

    1. Philox4×64-7 is a counter-based PRNG that supports one *stream* per seed. To seed two processes based on the seed "seed" and this PRNG, an application can—

        - take the SHA2-256 hash of "seed-mysimulation" as a new seed,
        - initialize the first process's PRNG with the new seed and a counter of 0, and
        - initialize the second process's PRNG with 1 plus the new seed and a counter of 0.

2. Some *dynamic threading* (*task-parallel*) platforms employ task schedulers where *tasks* or *subtasks* (sometimes called *strands* or *fibers*) are not assigned to a particular operating system process or thread. To ensure reproducible "randomness" in these platforms, PRNGs have to be assigned to tasks (rather than system processes or threads) and are not shared between tasks, and each task's PRNG can be initialized as given in the "general case" steps above (where the task's unique number is also known as a *pedigree*) (Leierson et al., 2012)[^10].

9 Existing RNG APIs in Programming Languages

As much as possible, **applications should use existing libraries and techniques** for cryptographic and high-quality RNGs. The following table lists application programming interfaces (APIs) for such RNGs for popular programming languages.

- PRNGs mentioned in the "High-Quality" column need to be initialized with a seed (see "**Seed Generation for Noncryptographic PRNGs**").
- The mention of a third-party library in this section does not imply that the library is the best one available for any particular purpose. The list is not comprehensive.
- See also **Paragon's blog post** on existing cryptographic RNGs.

Language	Cryptographic	High-Quality
.NET (incl. C# and VB.NET) (H)	`RandomNumberGenerator.Create()` in `System.Security.Cryptography` namespace; **airbreather/AirBreather.Common library** (CryptographicRandomGenerator)	`XoshiroPRNG.Net` package (XoRoShiRo128starstar, XoShiRo256plus, XoShiRo256starstar); `Data.HashFunction.MurmurHash` or `Data.HashFunction.CityHash` package (hash the string `seed + "_" + counter`)
C/C++ (G)	(C)	**`xoroshiro128plusplus.c`**; **`xoshiro256starstar.c`** **ihaque/xorshift** library (default seed uses `os.urandom()`); **`numpy.random.Generator`** with `Philox` or `SFC64` (since ver. 1.7); `hashlib.md5(b"%d_%d" % (seed, counter)).digest()`, `hashlib.sha1(b"%d_%d" % (seed, counter)).digest()`
Python (A)	`secrets.SystemRandom` (since Python 3.6); `os.urandom()`	
Java (A) (D)	(C); `java.security.SecureRandom` (F)	**`it.unimi.dsi/dsiutils` artifact** (XoRoShiRo128PlusPlusRandom, XoRoShiRo128StarStarRandom, XoShiRo256StarStarRandom, XorShift1024StarPhiRandom); **`org.apache.commons/commons-rng-simple`** artifact (`RandomSource` of `SFC_64`, `XO_RO_SHI_RO_128_PP`, `XO_RO_SHI_RO_128_SS`, `XO_SHI_RO_256_PP`, or `XO_SHI_RO_256_SS`)
JavaScript (B)	`crypto.randomBytes(byteCount)` (node.js only); `random-number-csprng` package (node.js only); `crypto.getRandomValues()` (Web)	`xoroshiro128starstar` package; `md5` package (`md5(seed+"_"+counter, {asBytes: true})`); `murmurhash3js` package (`murmurhash3js.x86.hash32(seed+"_"+counter)`); `crypto.createHash("sha1")` (node.js only)
Ruby (A) (E)	(C); `SecureRandom.rand()` (0 or greater and less than 1) (E); `SecureRandom.rand(N)` (integer) (E) (for both, `require 'securerandom'`);	`Digest::MD5.digest("#{seed}_#{counter}")`, `Digest::SHA1.digest("#{seed}_#{counter}")` (for both, `require 'digest'`)

	`sysrandom` gem	
PHP (A)	`random_int()`, `random_bytes()` (both since PHP 7)	`md5($seed.'_'.$counter, true);` `sha1($seed.'_'.$counter, true)`
Go	`crypto/rand` package	`md5.Sum` in `crypto/md5` package or `sha1.Sum` in `crypto/sha1` package (for both, hash the byte array `seed + "_" + counter`)
Rust	(C)	`rand_xoshiro` crate (Xoroshiro128PlusPlus, Xoshiro256PlusPlus, Xoshiro256StarStar, Xoshiro512StarStar)
Perl	`Crypt::URandom` module	`Crypt::Digest::MD5` module (`md5($seed.'_'.$counter)`); `Digest::SHA` module (`sha1($seed.'_'.$counter)`); `Digest::MurmurHash3` module (`murmurhash3($seed.'_'.$counter)`)
Other Languages	(C)	Hash the string `seed + "_" + counter` with MurmurHash3, xxHash64, CityHash, MD5, or SHA-1

- (A) The general RNGs of recent versions of Python and Ruby implement **Mersenne Twister**, which is not preferred for a high-quality RNG. PHP's `mt_rand()` implements or implemented a flawed version of Mersenne Twister.

- (B) JavaScript's `Math.random()` (which ranges 0 or greater and less than 1) is implemented using `xorshift128+` (or a variant) in the V8 engine, Firefox, and certain other modern browsers as of late 2017; `Math.random()` uses an "implementation-dependent algorithm or strategy", though (see ECMAScript sec. 20.2.2.27).

- (C) A cryptographic RNG implementation can—

    - read from the `/dev/urandom` device in Linux-based systems (using the `open` and `read` system calls where available)[^21],
    - call the `arc4random` or `arc4random_buf` method on FreeBSD or macOS,
    - call the `getentropy` method on OpenBSD, or
    - call the `BCryptGenRandom` API in Windows 7 and later,

  and only use other techniques if the existing ones are inadequate for the application. But unfortunately, resource-constrained devices ("embedded" devices) are much less likely to have a cryptographic RNG available compared to general-purpose computing devices such as desktop computers and smartphones (Wetzels 2017)[^22], although methods exist for implementing a cryptographic RNG on the Arduino (Peng 2017)[^23].

- (D) Java's `java.util.Random` class uses a 48-bit seed, so is not considered a high-quality RNG. However, a subclass of `java.util.Random` might be implemented as a high-quality RNG.

- (E) Ruby's `SecureRandom.rand` method presents a beautiful and simple API for generating numbers at random, in my opinion. Namely, `rand()` returns a number 0 or greater and less than 1, and `rand(N)` returns an integer 0 or greater and less than N.

- (F) In Java 8 and later, use `SecureRandom.getInstanceStrong()`. In Java earlier than 8, call `SecureRandom.getInstance("NativePRNGNonBlocking")` or, if that fails, `SecureRandom.getInstance("NativePRNG")`. For Android, especially versions 4.3 and earlier, see (Klyubin 2013)[^24]. Using the `SecureRandom` implementation `"SHA1PRNG"` is not recommended, because of weaknesses in seeding and RNG quality in implementations as of 2013 (Michaelis et al., 2013)[^25].

- (G) **`std::random_device`** was introduced in C++11, but its specification leaves considerably much to be desired. For example, `std::random_device` can fall back to a PRNG of unspecified quality without much warning. At best, `std::random_device` should not be used except to supplement other techniques for

generating random-behaving numbers.

- (H) The .NET Framework's `System.Random` class uses a seed of at most 32 bits, so is not considered a high-quality RNG. However, a subclass of `System.Random` might be implemented as a high-quality RNG.

10 Hash Functions

A *hash function* is a function that takes an arbitrary input of any size (such as an array of 8-bit bytes or a sequence of characters) and returns an output with a fixed number of bits. That output is also known as a *hash code*.

For pseudorandom number generation purposes:

- The individual bits of a hash code can serve as pseudorandom numbers, or the hash code can serve as the seed for a PRNG.
- Good hash functions include cryptographic hash functions (e.g., SHA2-256, BLAKE2) and other hash functions that tend to produce wildly dispersed hash codes for nearby inputs.
- Poor hash functions include linear PRNGs such as LCGs and the Xorshift family.

The use of hash functions for other purposes (such as data lookup and data integrity) is beyond the scope of this document. See my note on **hash functions**.

10.1 Procedural Noise Functions

Noise is a randomized variation in images, sound, and other data.[^26]

A *noise function* is similar to a hash function; it takes an *n*-dimensional point and, optionally, additional data, and outputs a pseudorandom number.[^27] Noise functions generate ***procedural noise*** such as **cellular noise**, **value noise**, and **gradient noise** (including **Perlin noise**). If the noise function takes additional data, that data—

- should include randomly generated or pseudorandom numbers, and
- should not vary from one run to the next while the noise function is used for a given purpose (e.g., to generate terrain for a given map).

10.2 Pseudorandom Functions

A *pseudorandom function* is a kind of hash function that takes—

- a *secret* (such as a password or a long-term key), and
- additional data such as a *salt* (which is designed to mitigate precomputation attacks) or a *nonce*,

and outputs a pseudorandom number. (If the output is encryption keys, the function is also called a *key derivation function*; see NIST SP 800-108.) Some pseudorandom functions deliberately take time to compute their output; these are designed above all for cases in which the secret is a password or is otherwise easy to guess — examples of such functions include PBKDF2 (RFC 2898), `scrypt` (RFC 7914), and Ethash. Pseudorandom functions are also used in proofs of work such as the one described in RFC 8019 sec. 4.4.

11 RNG Topics

This section discusses several important points on the use and selection of RNGs, including things to consider when shuffling or generating "unique" random identifiers.

11.1 Shuffling

In a list with `N` different items, there are `N` factorial (that is, `1 * 2 * ... * N`, or `N!`) ways to arrange the items in that list. These ways are called *permutations*[^28].

In practice, an application can **shuffle a list** by doing a **Fisher–Yates shuffle**, which is unfortunately easy to mess up — see (Atwood 2007)[^29] — and is implemented correctly in **another document of mine**.

However, if a PRNG admits fewer seeds (and thus can produce fewer number sequences) than the number of permutations, then there are **some permutations that that PRNG can't choose** when it shuffles that list. (This is not the same as *generating* all permutations of a list, which, for a list big enough, can't be done by any computer in a reasonable time.)

On the other hand, for a list big enough, it's generally **more important to have shuffles act random** than to choose from among all permutations.

An application that shuffles a list can do the shuffling—

1. using a cryptographic RNG, preferably one with a security strength of `b` bits or greater, or
2. if a noncryptographic RNG is otherwise appropriate, using a ***high-quality PRNG*** that—
    - has a `b`-bit or bigger state, and
    - is initialized with a seed derived from data with at least **`b` bits of *entropy***, or "randomness".

For shuffling purposes, `b` can usually be calculated by taking `n` factorial minus 1 (where `n` is the list's size) and calculating its bit length. A Python example is `b = (math.factorial(n)-1).bit_length()`. See also (van Staveren 2000, "Lack of randomness")[^30]. For shuffling purposes, an application may limit `b` to 256 or greater, in cases when variety of permutations is not important. For other sampling tasks, the following Python examples show how to calculate `b`:

- Choosing `k` out of `n` different items at random, in random order: `b = ((math.factorial(n)/math.factorial(n-k))-1).bit_length()`.
- Choosing `k` out of `n` different items at random, without caring about order (RFC 3797, sec. 3.3): `b = ((math.factorial(n)/(math.factorial(k) * math.factorial(n-k)))-1).bit_length()`.
- Shuffling `d` identical lists of `c` items: `b = ((math.factorial(d*c)/(math.factorial(d)**c))-1).bit_length()`.

11.2 Unique Random Identifiers

Some applications require generating unique identifiers, especially to identify database records or other shared resources. Examples of unique values include auto-incremented numbers, sequentially assigned numbers, primary keys of a database table, and combinations of these. Applications have also generated unique values at random.

The following are some questions to consider when generating unique identifiers:

1. Can the application easily check identifiers for uniqueness within the desired scope

and range (e.g., check whether a file or database record with that identifier already exists)[^31]?
2. Can the application tolerate the risk of generating the same identifier for different resources[^32]?
3. Do identifiers have to be hard to guess, be simply "random-looking", or be neither?
4. Do identifiers have to be typed in or otherwise relayed by end users[^33]?
5. Is the resource an identifier identifies available to anyone who knows that identifier (even without being logged in or authorized in some way)?[^34]
6. Do identifiers have to be memorable?

Some applications may also care about "unique random" values. Generally, however, values that are both *unique* and *random* are impossible. Thus, applications that want "unique random" values have to either settle for numbers that merely "look random"; or check for or tolerate possible duplicates; or pair randomly generated numbers with unique ones.

If the application can settle for "random-looking" unique integers:

- The application can produce a unique N-bit integer and pass that integer to a function that maps N-bit integers to N-bit integers in a reversible way (also called a *mixing function* with reversible operations; see "**Hash functions**" by B. Mulvey). This includes using the unique integer as the seed for a "full-period" linear PRNG, that is, a linear PRNG that goes through all N-bit integers exactly once before repeating[^35].
- The application can generate unique integers greater than 0 and less than K as follows:
  1. Set U to 0, and choose F, an N-bit function described earlier, where N is the number of bits needed to store the number K-minus-1.
  2. Calculate $F(U)$ then add 1 to U. If the result of F is less than K, output that result; otherwise, repeat this step.
  3. Repeat the previous step as needed to generate additional unique integers.

An application that generates unique identifiers should do so as follows:

- If the application can answer yes to question 1 or 2 above:
  - And yes to question 5: Generate a 128-bit-long or longer random integer using a cryptographic RNG.
  - And no to question 5: Generate a 32-bit-long or longer random integer using a cryptographic RNG.
- Otherwise:
  - If identifiers don't have to be hard to guess: Use a unique integer (either one that's naturally unique, or a randomly generated number that was checked for uniqueness).
  - If they do have to be hard to guess: Use a unique integer which is followed by a random integer generated using a cryptographic RNG (the random integer's length depends on the answer to question 5, as above).

This section doesn't discuss how to format a unique value into a text string (such as a hexadecimal or alphanumeric string), because ultimately, doing so is the same as mapping unique values one-to-one with formatted strings (which will likewise be unique).

11.3 Verifiable Random Numbers

Verifiable random numbers are randomly generated numbers (such as seeds for PRNGs) that are disclosed along with all the information necessary to verify their generation. Usually, such information includes randomly generated values and/or uncertain data to be determined and publicly disclosed in the future. Techniques to generate *verifiable random*

numbers (as opposed to cryptographic RNGs alone) are used whenever one party alone can't be trusted to produce a number at random. *Verifiable random numbers* that are disclosed *publicly* should not be used as encryption keys or other secret parameters.

Examples:

1. Generating verifiable randomness has been described in **RFC 3797**, which describes the selection process for the Nominations Committee (NomCom) of the Internet Engineering Task Force.
2. *Verifiable delay functions* calculate an output as well as a proof that the output was correctly calculated; these functions deliberately take much more time to calculate the output (e.g., to generate a random-behaving number from public data) than to verify its correctness.[^36] In many cases, such a function deliberately takes much more time than the time allowed to contribute randomness to that function.[^37]
3. In a so-called ***commitment scheme***, one computer generates data to be committed (e.g. a randomly generated number or a chess move), then reveals its hash code or digital signature (*commitment*), and only later reveals to all participants the committed data (along with other information needed, if any, to verify that the data wasn't changed in between). Examples of commitment schemes are *hash-based commitments*.[^37]
4. So-called *mental card game* (*mental poker*) schemes can be used in networked games where a deck of cards has to be shuffled and dealt to players, so that the identity of some cards is known to some but not all players.[^37]

12 Guidelines for New RNG APIs

This section contains guidelines for those seeking to implement RNGs designed for wide reuse (such as in a programming language's standard library). *As mentioned earlier, an application should use existing RNG implementations whenever possible.*

This section contains suggested requirements on cryptographic and high-quality RNGs that a new programming language can choose to adopt.

12.1 Cryptographic RNGs: Requirements

A cryptographic RNG generates random bits that behave like independent uniform random bits, such that an outside party has no more than negligible advantage in correctly guessing prior or future unseen output bits of that RNG even after knowing how the RNG works and/or extremely many outputs of the RNG, or prior unseen output bits of that RNG after compromising its security, such as reading its internal state.[^38]

If a cryptographic RNG implementation uses a PRNG:

- Let S be the security strength of the RNG. S is at least 128 bits and should be at least 256 bits.
- Before the RNG generates a pseudorandom number, the RNG has to have been initialized to a state that ultimately derives from data that, as a whole, is at least as hard to guess as an ideal process of generating S many independent uniform random bits[^39].

A cryptographic RNG is not required to reseed itself.

Examples: The following are examples of cryptographic RNGs:

- Randomness extractors or cryptographic **hash functions** that take very hard-to-predict signals from two or more **nondeterministic sources** as input.
- A "fast-key-erasure" generator described by D.J. Bernstein in his blog (Bernstein 2017)[^40].
- The `Hash_DRBG` and `HMAC_DRBG` generators specified in NIST SP 800-90A. The SP 800-90 series goes into further detail on how RNGs appropriate for information security can be constructed, and inspired much of this section.
- An RNG made up of two or more independently initialized cryptographic RNGs of different designs.[^41]
- RFC 8937 describes an RNG that hashes another cryptographic RNG's output with a secret value derived from a long-term key.

12.2 High-Quality RNGs: Requirements

A PRNG is a high-quality RNG if—

- it generates bits that behave like independent uniform random bits (at least for nearly all practical purposes outside of information security),
- the number of different seeds the PRNG admits without shortening or compressing those seeds is 2^{63} or more (that is, the PRNG can produce any of at least 2^{63} different number sequences, which it can generally do only if the PRNG has at least 63 bits of state), and
- it either—
  - provides multiple sequences that are different for each seed, have at least 2^{64} numbers each, do not overlap, and behave like independent sequences of numbers (at least for nearly all practical purposes outside of information security),
  - has a maximum "random" number cycle length equal to the number of different seeds the PRNG admits, or
  - has a minimum "random" number cycle length of 2^{127} or greater.

Every cryptographic RNG is also a high-quality RNG.

Where a noncryptographic PRNG is appropriate, an application should use, if possible, a high-quality PRNG that admits any of 2^{127} or more seeds. (This is a recommendation, since as stated above, high-quality PRNGs are required to admit only 2^{63} or more seeds.)

> **Examples:** Examples of high-quality PRNGs include xoshiro256**, xoroshiro128**, xoroshiro128++, Philox4×64-7, and SFC64. I give additional examples in a **separate page**.

12.3 Designs for PRNGs

The following are some ways a PRNG can be implemented:

- As a *stateful* object that stores an internal state and transforms it each time a "random" number is generated. This kind of PRNG is initialized by converting a *seed* to an internal state.
- As a (stateless) function that transforms an internal state and outputs "random" numbers and the transformed state. This design is often seen in Haskell and other functional programming languages.
- As a (stateless) "splittable PRNG", further described in my document on **testing PRNGs**.

12.4 Implementing New RNG APIs

A **programming language API** designed for reuse by applications could implement RNGs using the following guidelines:

1. The RNG API can include a method that fills one or more memory units (such as 8-bit bytes) completely with random bits. See example 1.
2. If the API implements an automatically-seeded RNG, it should not allow applications to initialize that same RNG with a seed for reproducible "randomness"[^42] (it may provide a separate PRNG to accept such a seed). See example 2.
3. If the API provides a PRNG that an application can seed for reproducible "randomness", it should document that PRNG and any methods the API provides that use that PRNG (such as shuffling and Gaussian number generation), and should not change that PRNG or those methods in a way that would change the "random" numbers they deliver for a given seed. See example 2.
4. A new programming language's **standard library** ought to include the following methods for generating numbers that behave like independent uniformly distributed numbers (see my document on **randomization and sampling methods** for details).
    - Four methods for integers: 0 to `n` including `n`, 0 to `n` excluding `n`, `a` to `b` including `b`, and `a` to `b` excluding `b`.
    - A method to sample real numbers from the open interval `(a, b)`.

Examples:

1. A C language RNG method for filling memory could look like the following: `int random(uint8_t[] bytes, size_t size);`, where `bytes` is a pointer to an array of 8-bit bytes, and `size` is the number of random 8-bit bytes to generate, and where 0 is returned if the method succeeds and nonzero otherwise.
2. A Java API that follows these guidelines can contain two classes: a `RandomGen` class that implements an unspecified but general-purpose RNG, and a `RandomStable` class that implements an SFC64 PRNG that is documented and will not change in the future. `RandomStable` includes a constructor that takes a seed for reproducible "randomness", while `RandomGen` does not. Both classes include methods described in point 4, but `RandomStable` specifies the exact algorithms to those methods and `RandomGen` does not. At any time in the future, `RandomGen` can change its implementation to use a different RNG while remaining backward compatible, while `RandomStable` has to use the same algorithms for all time to remain backward compatible, especially because it takes a seed for reproducible "randomness".

13 Acknowledgments

I acknowledge—

- the commenters to the CodeProject version of this page (as well as a similar article of mine on CodeProject), including "Cryptonite" and member 3027120,
- Sebastiano Vigna,
- Severin Pappadeux, and
- Lee Daniel Crocker, who reviewed this document and gave comments.

14 Notes

[^1]: See also the question titled "Matlab rand and c++ rand()" on *Stack Overflow*.

[^2]: A distinction between *cryptographic* and *noncryptographic* RNGs seems natural, because many programming languages offer a general-purpose RNG (such as C's `rand` or Java's `java.util.Random`) and sometimes an RNG intended for information security purposes (such as Ruby's `SecureRandom`).

[^3]: For example, see F. Dörre and V. Klebanov, "Practical Detection of Entropy Loss in Pseudo-Random Number Generators", 2016.

[^4]: Items that produce numbers or signals that follow a non-uniform distribution are not considered RNGs in this document. (For example, Gaussian and similar noise generators are not considered RNGs.) Many of these items, however, typically serve as sources from which uniform random-behaving integers can be derived through *randomness extraction* techniques (see "**Seed Generation**").
Likewise, items that produce floating-point numbers are not considered RNGs here, even if they sample from a uniform distribution. An example is the dSFMT algorithm, which ultimately uses a generator of pseudorandom integers.

[^5]: See also the FIPS 200 definition ("The protection of information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction in order to provide confidentiality, integrity, and availability") and ISO/IEC 27000.

[^6]: However, some versions of GLSL (notably GLSL ES 1.0, as used by WebGL 1.0) might support integers with a restricted range (as low as -1024 to 1024) rather than 32-bit or bigger integers as are otherwise common, making it difficult to write hash functions for generating pseudorandom numbers. An application ought to choose hash functions that deliver acceptable "random" numbers regardless of the kinds of numbers supported.

An alternative for GLSL and other fragment or pixel shaders to support randomness is to have the shader sample a "noise texture" with random data in each pixel; for example, C. Peters, "**Free blue noise textures**", *Moments in Graphics*, Dec. 22, 2016, discusses how so-called "blue noise" can be sampled this way.

See also N. Reed, "Quick And Easy GPU Random Numbers In D3D11", Nathan Reed's coding blog, Jan. 12, 2013.

[^7]: For more information, see "**Floating-Point Determinism**" by Bruce Dawson, the white paper "**Floating Point and IEEE 754 Compliance for NVIDIA GPUs**", and an **Intel webinar**.

[^8]: For integers, this problem also occurs, but is generally limited to the question of rounding after an integer division or remainder, which different programming languages answer differently.

[^9]: *Fixed-point numbers* are integers that store multiples of 1/*n* (e.g. 1/10000, 1/256, or 1/65536). Their resolution doesn't vary depending on the number, unlike with floating-point numbers. "**The Butterfly Effect - Deterministic Physics in The Incredible Machine and Contraption Maker**" is one use case showing how fixed-point numbers aid reproducibility. I have written a sample **Python implementation** of fixed-point numbers.

[^10]: Leierson, C.E., et al., "Deterministic Parallel Random-Number Generation for Dynamic Multithreading Platforms", 2012.

[^11]: Müller, S. "CPU Time Jitter Based Non-Physical True Random Number Generator".

[^12]: Liebow-Feeser, J., "Randomness 101: LavaRand in Production", blog.cloudflare.com, Nov. 6, 2017.

[^13]: Liebow-Feeser, J., "LavaRand in Production: The Nitty-Gritty Technical Details", blog.cloudflare.com, Nov. 6, 2017.

[^14]: Rather than generating a seed, these steps could be a way to simulate a source of numbers chosen independently and uniformly at random. However, this is generally slower than using PRNGs to simulate that source.

[^15]: For example, many questions on *Stack Overflow* highlight the pitfalls of creating a new instance of the .NET Framework's `System.Random` each time pseudorandom numbers are needed, rather than only once in the application. See also Johansen, R. S., "**A Primer on Repeatable Random Numbers**", Unity Blog, Jan. 7, 2015.

[^16]: Salmon, John K., Mark A. Moraes, Ron O. Dror, and David E. Shaw. "Parallel random numbers: as easy as 1, 2, 3." In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-12. 2011.

[^17]: P. L'Ecuyer, D. Munger, et al. "Random Numbers for Parallel Computers: Requirements and Methods, With Emphasis on GPUs", April 17, 2015, section 4, goes in greater detail on ways to initialize PRNGs for generating pseudorandom numbers in parallel, including how to ensure reproducible "randomness" this way if that is desired.

[^18]: For single-cycle PRNGs, the probability of overlap for N processes each generating L numbers with a PRNG whose cycle length is P is at most N*N*L/P (S. Vigna, "On the probability of overlap of random subsequences of pseudorandom number generators", *Information Processing Letters* 158 (2020)). Using two or more PRNG designs can reduce correlation risks due to a particular PRNG's design. For further discussion and an example of a PRNG combining two different PRNG designs, see Agner Fog, "**Pseudo-Random Number Generators for Vector Processors and Multicore Processors**", *Journal of Modern Applied Statistical Methods* 14(1), article 23 (2015).

[^19]: Bauke and Mertens, "Random numbers for large-scale distributed Monte Carlo simulations", 2007.

[^20]: Besides the seed, other things are hashed that together serve as a *domain separation tag* (see, e.g., the work-in-progress document "draft-irtf-cfrg-hash-to-curve"). Note the following: - In general, hash functions carry the risk that two processes will end up with the same PRNG seed (a *collision risk*) or that a seed not allowed by the PRNG is produced (a "rejection risk"), but this risk decreases the more seeds the PRNG admits (see "**Birthday problem**"). - M. O'Neill (in "Developing a `seed_seq` Alternative", Apr. 30, 2015) developed hash functions (`seed_seq_fe`) that are designed to avoid collisions if possible, and otherwise to reduce collision bias. For example, `seed_seq_fe128` hashes 128-bit seeds to 128-bit or longer unique values. - An application can handle a rejected seed by hashing with a different value or by using a backup seed instead, depending on how tolerant the application is to bias. - See also Matsumoto, M., et al., "Common defects in initialization of pseudorandom number generators", *ACM Transactions on Modeling and Computer Simulation* 17(4), Sep. 2007.

[^21]: Using the similar `/dev/random` is not recommended, since in some implementations it can block for seconds at a time, especially if not enough randomness is available. See also **"Myths about /dev/urandom"**.

[^22]: Wetzels, J., "33C3: Analyzing Embedded Operating System Random Number Generators", samvartaka.github.io, Jan. 3, 2017.

[^23]: B. Peng, "Two Fast Methods of Generating True Random Numbers on the Arduino", GitHub Gist, December 2017.

[^24]: A. Klyubin, "Some SecureRandom Thoughts", Android Developers Blog, Aug. 14,

2013.

[^25]: Michaelis, K., Meyer, C., and Schwenk, J. "Randomly Failed! The State of Randomness in Current Java Implementations", 2013.

[^26]: There are many kinds of noise, such as procedural noise (including Perlin noise, cellular noise, and value noise), **colored noise** (including white noise and pink noise), periodic noise, and noise following a Gaussian or other **probability distribution**. See also two articles by Red Blob Games: **"Noise Functions and Map Generation"** and **"Making maps from noise functions"**.

[^27]: Noise functions include functions that combine several outputs of a noise function, including by **fractional Brownian motion**. By definition, noise functions deliver the same output for the same input.

[^28]: More generally, a list has `N! / (W_1! * W_2! * ... * W_K!)` permutations (a **multinomial coefficient**), where `N` is the list's size, `K` is the number of different items in the list, and `W_i` is the number of times the item identified by `i` appears in the list. However, this number is never more than `N!` and suggests using less randomness, so an application need not use this more complicated formula and may assume that a list has `N!` permutations even if some of its items occur more than once.

[^29]: Atwood, Jeff. "**The danger of naïveté**", Dec. 7, 2007.

[^30]: van Staveren, Hans. **"Big Deal: A new program for dealing bridge hands"**, Sep. 8, 2000

[^31]: For applications distributed across multiple computers (e.g., servers), this check is made easier if each computer is assigned a unique value from a central database, because then the computer can use that unique value as part of unique identifiers it generates and ensure that the identifiers are unique across the application without further contacting other computers or the central database. An example is Twitter's **Snowflake service**.

[^32]: In theory, generating two or more random integers of the same size runs the risk of producing a duplicate number this way. However, this risk decreases as that size increases (see "**Birthday problem**"). For example, in theory, an application has a 50% chance for duplicate numbers after generating— - about 2.7 billion billion random 122-bit integers (including those found in version-4 UUIDs, or universally unique identifiers), - about 1.4 million billion billion random 160-bit integers, or - about 93 billion billion billion random 192-bit integers.

[^33]: If an application expects end users to type in a unique identifier, it could find that very long unique identifiers are unsuitable for it (e.g. 128-bit numbers take up 32 base-16 characters). There are ways to deal with these and other long identifiers, including (1) separating memorable chunks of the identifier with a hyphen, space, or another character (e.g., "ABCDEF" becomes "ABC-DEF"); (2) generating the identifier from a sequence of memorable words (as in Electrum or in Bitcoin's BIP39); or (3) adding a so-called "checksum digit" at the end of the identifier to guard against typing mistakes. The application ought to consider trying (1) or (2) before deciding to use shorter identifiers than what this document recommends.

[^34]: Note that the *insecure direct object references* problem can occur if an application enables access to a sensitive resource via an easy-to-guess identifier, but without any access control checks.

[^35]: "Full-period" linear PRNGs include so-called **linear congruential generators** with a power-of-two modulus. For examples of those, see tables 3, 5, 7, and 8 of Steele and

Vigna, "**Computationally easy, spectrally good multipliers for congruential pseudorandom number generators**", arXiv:2001.05304 [cs.DS].

[^36]: Verifiable delay functions are different from proofs of work, in which there can be multiple correct answers. These functions were first formally defined in Boneh, D., Bonneau, J., et al., "Verifiable Delay Functions", 2018, but such functions appeared earlier in Lenstra, A.K., Wesolowski, B., "A random zoo: sloth, unicorn, and trx", 2015.

[^37]: It is outside the scope of this page to explain how to build a protocol using verifiable delay functions, commitment schemes, or mental card game schemes, especially because such protocols are not yet standardized for general use and few implementations of them are used in production.

[^38]: Implementing a cryptographic RNG involves many security considerations, including these: 1. If an application runs code from untrusted sources in the same operating system process in which a cryptographic RNG's state is stored, it's possible for malicious code to read out that state via side-channel attacks. A cryptographic RNG should not be implemented in such a process. See (A) and see also (B). 2. A cryptographic RNG's state could be reused due to process forking or virtual machine snapshot resets. See (C) and (D), for example. 3. If a cryptographic RNG is not "constant-time" (the RNG is data-dependent), its timing differences could be exploited in a security attack.

`(A) "Post-Spectre Threat Model Re-Think" in the Chromium source code repository (May 29, 2018).
(B) Bernstein, D.J. "Entropy Attacks!", Feb. 5, 2014.
(C) Everspaugh, A., Zhai, Y., et al. "Not-So-Random Numbers in Virtualized Linux and the Whirlwind RNG", 2014.
(D) Ristenpart, T., Yilek, S. "When Good Randomness Goes Bad: Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography", 2010.
For a detailed notion of a secure RNG, see Coretti, Dodis, et al., "Seedless Fruit is the Sweetest: Random Number Generation, Revisited", 2019.`

[^39]: This data can come from nondeterministic sources, and also include process identifiers, time stamps, environment variables, pseudorandom numbers, virtual machine guest identifiers, and/or other data specific to the session or to the instance of the RNG. See also NIST SP 800-90A and the previous note.

[^40]: Bernstein, D.J. "Fast-key-erasure random number generators", Jun. 23, 2017.

[^41]: An example is the "shrinking generator" technique to combine two RNGs; see J. D. Cook, "Using one RNG to sample another", June 4, 2019, for more.

[^42]: Allowing applications to do so would hamper forward compatibility — the API would then be less free to change how the RNG is implemented in the future (e.g., to use a cryptographic or otherwise "better" RNG), or to make improvements or bug fixes in methods that use that RNG (such as shuffling and Gaussian number generation). (As a notable example, the V8 JavaScript engine recently changed its `Math.random()` implementation to use a variant of `xorshift128+`, which is backward compatible because nothing in JavaScript allows `Math.random()` to be seeded.) Nevertheless, APIs can still allow applications to provide additional input ("entropy") to the RNG in order to increase its randomness rather than to ensure repeatability.

15 License