International Standard

ISO/IEC 18031

Third edition
2025-02

Information technology — Security techniques — Random bit generation

Technologies de l'information — Techniques de sécurité — Génération de bits aléatoires

Reference number
ISO/IEC 18031:2025(en)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and https://patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC/JTC 1 *Information Technology*, Subcommittee SC 27, *Information security, cybersecurity and privacy protection*.

This third edition cancels and replaces the second edition (ISO/IEC 18031:2011), which has been technically revised. It also incorporates the Amendment ISO/IEC 18031:2011/Amd 1:2017 and the Technical Corrigendum ISO/IEC 18031:2011/Cor 1:2014.

The main changes are as follows:

- removal of the MQ_DRBG, Micali-Schnorr DRBG, Dual_EC_DRBG and SHA-1;
- addition and harmonization of the terms and definitions in Clause 3;
- addition of conversion methods for random number generation;
- update of the requirements for DRBGs and NRBGs.

Any feedback or questions on this document should be directed to the user’s national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction

This document sets out specific requirements that, when met, will result in the development of a random bit generator that can be applicable to cryptographic applications.

Numerous cryptographic applications involve the use of random bits. These cryptographic applications include the following:

- random keys and initialization values (*IVs*) for encryption,
- random keys for keyed MAC algorithms,
- random private keys for digital signature algorithms,
- random values to be used in entity authentication mechanisms,
- random values to be used in key-establishment protocols,
- random PINs and passwords,
- nonces.

The purpose of this document is to establish a conceptual model, terminology and requirements related to the building blocks and properties of systems used for random bit generation in or for cryptographic applications.

It is possible to categorize random bit generators into two types, namely, non-deterministic and deterministic random bit generators.

A non-deterministic random bit generator can be defined as a random bit generating mechanism that continuously uses a source of entropy to generate a random bit stream.

A deterministic random bit generator can be defined as a bit generating mechanism that uses deterministic mechanisms such as cryptographic algorithms to generate a random bit stream. In this type of bit stream generation, there is a specific input (normally called a seed) and perhaps some optional input, which, depending on its application, can either be publicly available or not. The seed is processed by a function which provides an output.

NOTE This document also discusses hybrid random bit generators, which incorporate elements of both non-deterministic and deterministic generators.

In this document, variable symbols and variable descriptive terms are given in italic font.

Information technology — Security techniques — Random bit generation

1 Scope

This document specifies a conceptual model for a random bit generator for cryptographic purposes, together with the elements of this model.

This document specifies the characteristics of the main elements required for both non-deterministic and deterministic random bit generators. It also establishes the security requirements for both non-deterministic and deterministic random bit generators.

Techniques for statistical testing of random bit generators for the purposes of independent verification or validation and detailed designs for such generators are outside the scope of this document.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 9797-2, *Information security — Message authentication codes (MACs) — Part 2: Mechanisms using a dedicated hash-function*

ISO/IEC 10118-3, *IT Security techniques — Hash-functions — Part 3: Dedicated hash-functions*

ISO/IEC 19790, *Information technology — Security techniques — Security requirements for cryptographic modules*

ISO/IEC 20543, *Information technology — Security techniques — Test and analysis methods for random bit generators within ISO/IEC 19790 and ISO/IEC 15408*

ISO/IEC 29192-5, *Information technology — Security techniques — Lightweight cryptography — Part 5: Hash-functions*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

3.1
algorithm
clearly specified mathematical process for the computation of a set of rules that, if followed, will give a prescribed result

3.2
backward secrecy
assurance that previous *random bit generator (RBG)* (3.40) output values cannot be determined with practical computational effort from knowledge of current or subsequent (future) output values

3.3
bit stream
continuous output of bits from a device or mechanism

3.4
bit-string
finite sequence of ones and zeroes

3.5
block cipher
symmetric encryption system with the property that the encryption *algorithm* (3.1) operates on a block of plaintext to yield a block of ciphertext

Note 1 to entry: The block ciphers standardized in ISO/IEC 18033-3 have the property that plaintext and ciphertext blocks are of the same length.

[SOURCE: ISO/IEC 18033-1:2021, 3.6]

3.6
conditioning
method of processing the data to reduce bias and/or ensure that the entropy rate of the output is no less than some specified amount

3.7
cryptographic boundary
explicitly defined perimeter that establishes the boundary of all components (i.e. a set of hardware, software, or firmware) of the cryptographic module

[SOURCE: ISO/IEC TS 30104:2015, 3.4]

3.8
deterministic algorithm
algorithm (3.1) that, when given a particular input, always produces the same output

3.9
deterministic random bit generator
DRBG
random bit generator (3.40) that produces a random-appearing sequence of bits by applying a *deterministic algorithm* (3.8) to a suitably random initial value called a seed and, possibly, some secondary inputs upon which the security of the random bit generator does not depend

Note 1 to entry: In particular, non-deterministic sources may also form part of these secondary inputs.

3.10
deterministic random bit generator boundary
DRBG boundary
conceptual boundary that is used to explain the operations of a *deterministic random bit generator (DRBG)* (3.9) and its interaction with and relation to other processes

3.11
enhanced backward secrecy
assurance that the knowledge of the current internal state of a *random bit generator* (3.40) does not allow an adversary to derive, with practical computational effort, knowledge about previous output values

Note 1 to entry: Another term often found in the literature that is similar to enhanced backward secrecy is backtracking resistance.

[SOURCE: ISO/IEC 20543:2019, 3.6 modified — Note 1 to entry replaced and commas added after “derive” and “effort”]

3.12
enhanced forward secrecy

assurance that it is not feasible to determine (future) output values after sufficient *entropy* (3.13) has been mixed into the internal state, given knowledge of the current and previous internal state

Note 1 to entry: *Deterministic random bit generators* (3.9) are unable to achieve enhanced forward secrecy without the insertion of sufficient fresh entropy at the end of the sentence. Unlike forward and backward secrecy as well as *enhanced backward secrecy* (3.11), enhanced forward secrecy rests entirely on the ability of a continuous reseeding process to supply as much entropy as required to make the prediction of future outputs infeasible.

Note 2 to entry: It is possible for a random bit generator to have enhanced forward secrecy but still expand entropy, i.e. output a bit-string that can, in principle, be significantly "compressed". For instance, it is possible to consider a random bit generator design with a random source that produces (at each invocation) a 128-bit random string R with an estimated 128 bits of min entropy, with a 512-bit internal state $S(n)$, an internal state transition function giving $S(n+1)$:= SHA3-512($S(n)||R$), and an output generation function applying SHAKE-256 on $S(n)||R$ with up to 1 024 bits of output per invocation.

Note 3 to entry: Another term often found in the literature that is similar to enhanced forward secrecy is prediction resistance.

Note 4 to entry: If insufficient entropy is mixed into the internal state, enhanced forward secrecy is not achieved, and it is possible that a compromise of the internal state is not cured by the additional entropy due to "iterative guessing attacks."

3.13
entropy

measure of the disorder, randomness or variability in a closed system

Note 1 to entry: The entropy of a random variable X is a mathematical measure of the amount of information provided by an observation of X.

3.14
entropy rate

assessed amount of *entropy* (3.13) in a *bit-string* (3.4) divided by the number of bits in the *bit-string* (3.4)

3.15
entropy source

combination of a noise source, health tests, and optional *conditioning* (3.6) that produce random *bit-strings* (3.4) for use by a *random bit generator* (3.40)

Note 1 to entry: Entropy sources can be physical or non-physical, depending on the noise source.

3.16
forward secrecy

assurance that the knowledge of subsequent (future) output values cannot be determined with practical computational effort from current or previous output values

Note 1 to entry: This definition is specific to the context of *random bit generators* (3.40). It should not be confused with "forward secrecy" defined in the ISO/IEC 11770 series for key management.

3.17
full entropy

entropy rate (3.14) that is practically close to 1

3.18
full entropy source

source that produces output with *full entropy* (3.17)

3.19
full forward secrecy

property of a *deterministic random bit generator (DRBG)* (3.9) in which sufficient *entropy* (3.13) is provided during every generation process to meet the security requirements for the security strength to be supported by the DRBG

3.20
glass box
idealized mechanism that accepts inputs and produces outputs and is designed such that an observer can determine exactly how the outputs are computed from the inputs

3.21
hash-function
function that maps strings of bits of variable (but usually upper-bounded) length to fixed-length strings of bits, satisfying the following three properties:

- for a given output, it is computationally infeasible to find an input that maps to this output;
- for a given input, it is computationally infeasible to find a second input that maps to the same output;
- it is computationally infeasible to find any two distinct inputs which map to the same output

Note 1 to entry: Computational feasibility depends on the specific security requirements and environment. Refer to ISO/IEC 10118-1:2016, Annex C.

[SOURCE: ISO/IEC 10118-1:2016, 3.4 modified – third bullet point added to the definition; "two properties" changed to "three properties".]

3.22
hybrid DRBG
hybrid deterministic random bit generator
deterministic random bit generator (DRBG) (3.9) that is capable of accepting external input values during its operation

3.23
hybrid NRBG
hybrid non-deterministic random bit generator
random bit generator (3.40) with non-deterministic input from a noise source and that uses complex, stateful post-processing (e.g. cryptographic processing)

Note 1 to entry: A hybrid *non-deterministic random bit generator (NRBG)* (3.30) can be physical or non-physical depending on its *entropy* (3.13) source.

3.24
independent and identically distributed
property of a family of random variables stating that they share the same distribution and are mutually independent

[SOURCE: ISO/IEC 20543:2019, 3.14]

3.25
initialization value
value used in defining the starting point of a cryptographic *algorithm* (3.1)

Note 1 to entry: Examples of cryptographic algorithms that use an initialization value include hash-functions and encryption algorithms.

3.26
Kerckhoffs's box
idealized cryptosystem where the design and public keys are known to an adversary, but in which there are secret keys and/or other private information that are not known to an adversary

3.27
known-answer test

method of testing a deterministic mechanism where a given input is processed by the mechanism, and the resulting output is then compared to a corresponding known value

Note 1 to entry: Known-answer testing of a deterministic mechanism may also include testing the integrity of the software that implements the deterministic mechanism. For example, if the software implementing the deterministic mechanism is digitally signed, then the signature can be recalculated and compared to the known signature value.

3.28
min-entropy

lower bound of *entropy* (3.13) that is useful in determining a worst-case estimate of sampled entropy

Note 1 to entry: The bit-string X (or more precisely, the corresponding random variable that models random bit-strings of this type) has min-entropy k, if k is the largest value such that $\Pr[X = x] \leq 2^{-k}$. That is, X contains k bits of min-entropy or randomness.

3.29
noise source

component of an entropy source that contains the non-deterministic, entropy-producing activity (e.g. thermal noise or hard-drive seek times)

Note 1 to entry: A noise source does not output digital data, unlike a physical or non-physical noise source.

3.30
non-deterministic random bit generator
NRBG

random bit generator (3.40) that samples one or multiple entropy sources and, if operating correctly, has an output that is expected to be unpredictable for attackers with unbounded computational capabilities

[SOURCE: ISO/IEC 20543:2019, 3.19, modified — “continuously” and “over short timescales” have been deleted; “one or” has been added.]

3.31
non-physical entropy source

entropy source in which *entropy* (3.13) is credited from one or more *non-physical noise sources* (3.33) e.g. random-access memory (RAM) content or thread number

3.32
non-physical noise source

noise source that exploits system data, peripheral data, or user interaction and outputs digital data

3.33
one-way function

function with the property that it is easy to compute the output for a given input but it is computationally infeasible to find an input that maps to a given output

[SOURCE: ISO/IEC 11770-3:2021, 3.30]

3.34
output generation function

function in a *random bit generator (RBG)* (3.40) that computes the output of the RBG from the internal state of the RBG

3.35
physical entropy source

entropy source in which *entropy* (3.13) is counted only from a *physical noise source* (3.36)

3.36
physical noise source

noise source (3.29) that exploits physical phenomena from dedicated hardware or physical experiments and produces digitized random data

3.37
protection boundary
physical or conceptual perimeter that defines the secure domain into which an attacker cannot observe or influence the process in a malicious way (according to a chosen threat model)

3.38
pure DRBG
pure deterministic random bit generator
deterministic random bit generator (3.9) whose only external input is an initial seed

3.39
pure NRBG
pure non-deterministic random bit generator
random bit generator (3.40) that takes its non-deterministic input from a noise source, and for which any post-processing is non-cryptographic or stateless cryptographic

3.40
random bit generator
RBG
device or *algorithm* (3.1) that outputs a sequence of bits that appears to be statistically independent and unbiased

3.41
randomness source
source of randomness for a *random bit generator* (3.40) that can be an entropy source, a *non-deterministic random bit generator* (3.30), or a *deterministic random bit generator* (3.9)

3.42
reseeding
specialized internal state transition function that updates the internal state in the event that a new seed value is supplied by either computing a new internal state from the current internal state and the new seed value, or by replacing the internal state based only on the new seed value

Note 1 to entry: The term reseeding is used in a variety of ways in the literature. In this document, reseeding refers to a mechanism that replaces the current value of the internal state by a fresh value, which can either (partially) depend on the current value, or not. Elsewhere, a distinction is sometimes made between reseeding and seed update. In such cases, the term reseeding is only used for mechanisms that replace the internal state by a new value that does not depend on the current value (essentially a new seeding process), and the term seed update is used for a mechanism that computes the new internal state as a function of its current value and other (usually non-deterministic) data (see 9.6, item 3).

3.43
secret parameter
input to the *random bit generator* (3.40) that provides additional randomness in the event of a failure or compromise of the randomness source

Note 1 to entry: In practice, the secret parameter is often a key.

Note 2 to entry: The secret parameter is only useful if it has sufficient randomness.

Note 3 to entry: The secret parameter is not the same as a seed.

3.44
security strength
number associated with the amount of work (i.e. the number of operations of some sort) that is required to break a cryptographic *algorithm* (3.1) or system

Note 1 to entry: If the security strength associated with an algorithm or system is n bits, then it is expected that (roughly) 2^n basic operations are required to break it.

3.45
seed
bit-string (3.4) that is used as input to initialize the internal state of a *deterministic random bit generator (DRBG)* (3.9)

Note 1 to entry: The seed will determine a portion of the state of the DRBG.

3.46
seedlife
period of time between initializing or reseeding the *deterministic random bit generator (DRBG)* (3.9) with one *seed* (3.45) and reseeding that DRBG with a different seed

3.47
seed material
data used to form a seed for input to a *deterministic random bit generator (DRBG)* (3.9)

Note 1 to entry: The term is often used to refer to the bit stream provided by a randomness source.

3.48
seed value
input *bit-string* (3.4) from a randomness source that provides *entropy* (3.13) for a *deterministic random bit generator* (3.9)

3.49
state
condition of a *random bit generator* (3.41) or any part thereof at a particular instant

3.50
stochastic model
partial mathematical description of a random bit generator based on at least a qualitative understanding of the noise source which, together with possibly some data gathered empirically for parameter estimation, allows the derivation of entropy claims from the noise source

Note 1 to entry: In the context of evaluating random bit generators, it is recommended but not required that the stochastic model describe the behaviour of the raw random bits. Subsequent post-processing can make it more difficult to make a convincing case that the stochastic model is in sufficient correspondence with the workings of the device to be modelled to support the entropy claims to be shown. For instance, a stochastic model applied to the output random numbers of a deterministic random bit generator will be essentially untestable statistically. This is because cryptographic post-processing can render even very low entropy data which is indistinguishable from random noise at realistic sample sizes, at least from the point of view of any adversary lacking a stochastic model of the raw random bits.

[SOURCE: ISO/IEC 20543:2019, 3.30 modified — "from the noise source" added to the definition; in Note 1 to entry, the last word “numbers” has been replaced by “bits”; the last sentence has been split into two.]

3.51
working state
subset of the internal state that is used by a *deterministic random bit generator* (3.9) mechanism to produce pseudorandom bits at a given point in time

4 Symbols

For the purposes of this document, the following symbols apply.

0^l	all-zero bit-string of length l
$\Pr[x]$	probability of occurrence of x
IV	initialization value
$\lceil X \rceil$	Ceiling: the smallest integer greater than or equal to X. For example, $\lceil 5 \rceil = 5$, and $\lceil 5{,}3 \rceil = 6$.
$X \oplus Y$	bitwise exclusive-or (also bitwise addition mod 2) of bit-strings X and Y of the same length
$X \mid\mid Y$	concatenation of two separate bit-strings X and Y in that order
$\mid a \mid$	the length in bits of string a
$x \bmod n$	The unique remainder r, $0 \le r \le n$-1, when integer x is divided by n. For example, 23 mod 7 = 2.

5 Properties and requirements of a random bit generator

5.1 Properties of arandom bit generator

The properties of randomness can be demonstrated by tossing a coin in the air and observing which side is uppermost when it lands, where one side is called “heads” (H) and the other is called “tails” (T). A coin also has a rim, but the probability that a coin can land on its rim is so unlikely an occurrence that, for the purpose of this demonstration, it can be ignored.

Flipping a coin multiple times produces an ordered series of coin flip results denoted as a series of H(s) and T(s). For example, the sequence “HTTHT” (reading left to right) indicates a head followed by a tail, followed by a tail, followed by a head, followed by a tail. This coin-flip sequence can be translated into a binary string in a straightforward manner by assigning H to a binary one ("1") and T to a binary zero ("0"); the resulting example bit-string is "10010".

The required properties of randomness can be examined using the example of the idealized coin toss described above. The result of each coin flip is:

a) Unpredictable: Before the flip, it is unknown whether the coin will land showing a head or a tail. Also, if that flip is kept secret, it is not possible to determine what the flip was if any subsequent flip outcome is known. The unpredictability after the flip depends on whether the observer can observe the coin flip or not. The notion of entropy quantifies the amount of unpredictability or uncertainty relative to an observer and will be discussed more thoroughly later in this document;

b) Unbiased: that is, each potential outcome has the same chance of occurring; and

c) Independent: the coin flip is said to be memoryless; whatever happened before the current flip does not influence it.

Such a series of idealized coin flips is directly applicable to a random bit generator (RBG). The RBGs specified in this document try to simulate a series of idealized coin flips.

As indicated above, unpredictability is a required property of an RBG. Predicting the output of a properly implemented and working RBG is not expected to be possible.

The decision whether to incorporate enhanced forward secrecy (which is an optional feature of an RBG) is determined by the needs of the consuming application. The following factors should be considered when deciding to incorporate enhanced forward secrecy:

1) An RBG without enhanced forward secrecy can be secure for a consuming application if exposure of the internal state of the RBG is unlikely or if any exposure is mitigated by replacement of the RBG. For example, a smart card may be initialized at the point of manufacture with sufficient entropy in the seed, and the smart card is set to expire after a limited time (e.g. two or three years). Compromise of a smart

card should be difficult and evident to the user. Therefore, a compromised smart card can be replaced with a new one.

2) An RBG should be capable of providing enhanced forward secrecy if the consuming application has identified a risk that the internal state can be exposed without detection. An example is a system which is compromised by a cyber attack that accessed the internal state of an RBG. If the compromise is not discovered or properly remediated, the system can continue performing cryptographic operations relying upon the RBG. Enhanced forward secrecy can enable the RBG to return to a secure internal state.

5.2 Requirements of an RBG

All RBGs specified in this document, both deterministic and non-deterministic, shall satisfy the requirements provided below in 1) to 12). These requirements are fundamental to the security of many cryptographic mechanisms that require random input and help prevent misuse in consuming applications.

The threshold between feasible and infeasible shall be determined by the overall requirement for the minimum acceptable strength of cryptographic security that is required by the application.

1) RBGs for general cryptographic applications shall output bit sequences that are indistinguishable from uniformly distributed and independently generated bit sequences under standard cryptographic assumptions. Suitable standard assumptions can be the hardness of breaking a block cipher or the difficulty of inverting a particular one-way function. The seeding process (and, optionally, any re-seeding that can happen during their operation) shall provide entropy that is commensurate with the intended security strength of the RBG, which shall be assessed using the evaluation methodology laid out in ISO/IEC 20543. See Annex G for RBG assurance.

2) Both the deterministic components and the entropy sources of RBGs shall be designed to resist side channel and fault attacks that are potentially feasible in the application environment.

3) From the perspective of an adversary, the internal state of the RBG shall always contain at least 128 bits of entropy and at least as much as the security strength of the consuming application requires. The perspective of an adversary includes all information that the adversary can effectively exploit, i.e. by means of classical computational attacks, quantum attacks, or relevant side channel and fault attacks.

4) The RBG shall not generate bits unless the generator has been assessed to possess sufficient entropy. The criteria for sufficiency shall be the greater of the requirements of this document and the requirements of the consuming application.

5) On detection of an error, the RBG shall either a) enter a permanent error state, or b) not output any random bits and be able to recover from a loss or compromise of entropy if the permanent error state is deemed unacceptable for the application requirements. These requirements may be satisfied procedurally or inherently in the design.

   NOTE 1 See 8.8 and 9.8 for NRBG and DRBG information on RBG errors and health tests.

6) The design and implementation of an RBG shall have a defined protection boundary. The protection boundary shall be as specified in ISO/IEC 19790 (see Annex H).

7) The probability that the RBG can “misbehave” in some pathological way that violates the output requirements (e.g. give an output that is constant or has a short cycle, i.e. looping such that the same output is repeated) shall be sufficiently small. This implies that the probability of error should be consistent with the overall confidence in correct operation that is required of the RBG, which is not necessarily the same as the required strength of cryptographic security.

8) The RBG design shall include methods to prohibit predictable influence, manipulation, or predicting the output of the RBG by observing the generator's physical characteristics (e.g. power consumption, timing, or emissions).

9) The implementation shall be designed to allow validation, including specific design assertions about what the RBG is not intended to do. The validation of an RBG means that the RBG behaves as expected, not just during normal operation but also at the boundaries of the intended operational conditions.

With respect to code-dependent RBGs, security-relevant branches in the code governing behaviour in exceptional conditions (e.g. initialization, failed health tests) shall be validated by deliberately forcing all error conditions to occur during validation testing.

10) There shall be design evidence (theoretical, empirical, or both) to support all security requirements for the RBG, including protection from misbehaviour.

11) An RBG shall have enhanced backward secrecy. This means that given all accessible information about the RBG (comprising some subset of inputs, algorithms and outputs), it shall be computationally infeasible (up to the specified security strength) to compute or otherwise determine any previous output bit.

12) An RBG shall have forward secrecy.

NOTE 2 Requirement 11 (enhanced backward secrecy) implies backward secrecy.

In some circumstances it is desirable to combine RBGs; if RBGs are combined then the method used shall be in accordance with one of the approaches specified in Annex A.

5.3 Additional information for an RBG

Additional information for an RBG is as follows.

1) If the RBG is capable of operating in more than one mode (e.g. operational mode or test mode), the RBG should, upon request, return information about the mode in which it is operating.

2) A consuming application may require that the RBG be run in a test mode, for example, when using a non-secret seed for generating bits for testing purposes only. When an RBG is in the test mode, the RBG shall not be capable of being used to generate (secret) operational output. The RBG shall not operate in the test mode using a seed to be used in the operational mode.

3) Considered as a glass box, an RBG may provide enhanced forward secrecy. If supported, it means that given all accessible information about the RBG (comprising some subset of inputs, algorithms, internal state, and outputs), it shall be infeasible (up to the specified security strength) to compute or predict any future output bit at the time that enhanced forward secrecy was requested.

6 RBG model

6.1 Conceptual functional model for random bit generation

Figure 1 depicts a conceptual functional model for random bit generation. This model and the associated requirements and objectives specify what RBGs are expected to achieve without constraining or mandating how the implementation shall be done, regardless of whether the RBG is non-deterministic or deterministic. Since not all of the important aspects of random bit generation can be algorithmically specified, this functional view of random bit generation is central to the definition of RBGs.

The functional model encompasses everything required to produce random bits. This holistic approach is necessary to ensure that RBG output will be as random as desired.

When an RBG product or system component does not directly incorporate all of the functional components or address all of the functional requirements specified in this document, that RBG can still conform to this document if the RBG component is used within a system that supplies the missing elements and satisfies the remaining requirements. Such an RBG is considered to possess residual functional requirements. Those residual requirements become a system prerequisite for use of the RBG.

NOTE Figure 1 shows that there are RBGs that do not incorporate all functional components. Physical NRBGs, for example, do not necessarily need additional inputs.

6.2 RBG basic components

6.2.1 Introduction to the RBG basic components

The model as shown in Figure 1 has six basic components; however, it can be possible to meet all the functional requirements without incorporating all the basic components. They are:

a) the randomness source;

b) additional inputs;

c) the internal state;

d) the internal state transition function;

e) the output generation function; and

f) health tests.

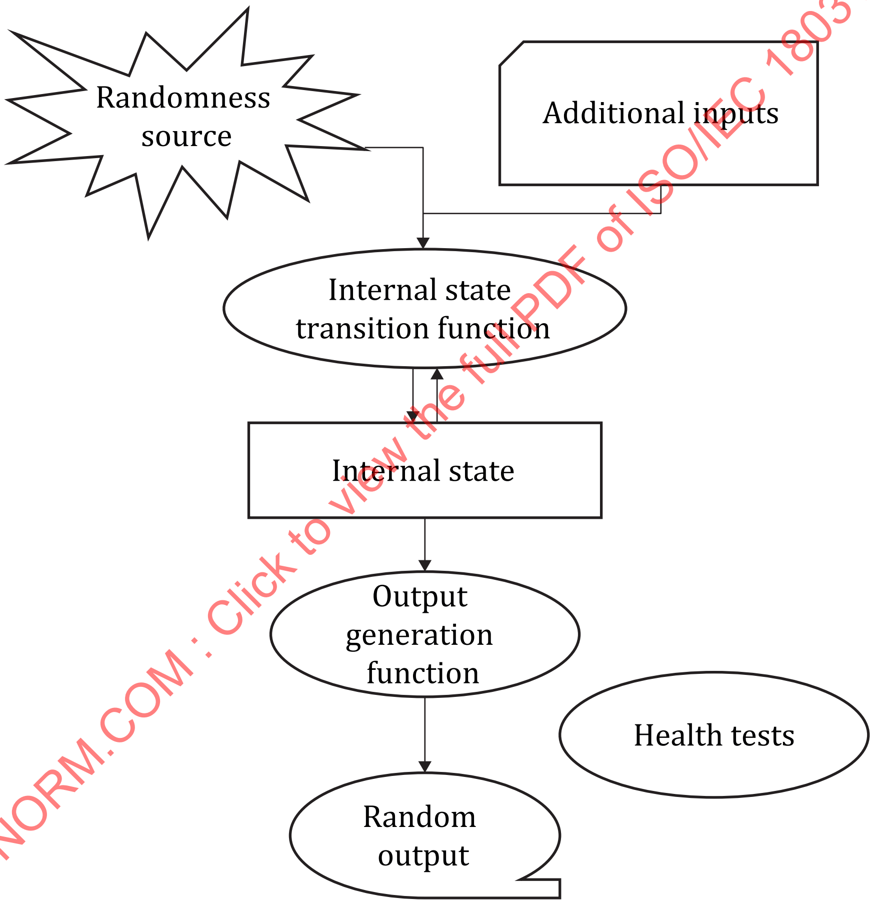

Figure 1 — RBG functional model

6.2.2 Randomness source

6.2.2.1 Randomness source overview

The randomness source is the source of unpredictable bits. These bits can be biased and in many cases, can be dependent on one another to some extent. In the case of a DRBG, the randomness source can be a distinct and possibly remote NRBG. However, for an NRBG, the randomness source component is an entropy source that has a noise source with some entropy-producing activity, a method for detecting this activity, and a digitization mechanism.

The randomness source shall produce bits with non-zero entropy. It is the only model component that produces entropy. For an NRBG, the entropy source encompasses everything that is not deterministic in the RBG model, along with whatever is required for the source to manifest itself in bits (as opposed to analogue signals or other non-digital activity). In a DRBG, the randomness source has traditionally been left unspecified, even though the unpredictability of the generator ultimately depends on the entropy of a special input called a seed. The model does not assume anything about the predictability, bias, or independence of the bits produced by the source. The only assumption is that the randomness source has non-zero entropy. However, all RBG designs are required to establish specific conditions that shall hold for the design to work.

6.2.2.2 Requirements for an entropy source

The requirements for the entropy source used by an RBG are as follows.

1) The entropy source shall be based upon well-established principles, or extensively characterized behaviour.

2) The entropy rate shall be assessable, or the collection shall be self-regulating, so that the amount of entropy per collection unit or event reliably achieves or exceeds a designed lower bound.

3) The entropy source shall be designed so that any manipulation (such as the ability to control the entropy source), influence (such as the ability to bias the entropy source), or its observation by some unauthorized external body shall be minimized according to the requirements of the RBG design.

4) Loss or severe degradation of an entropy source shall be detectable.

Methods for evaluating the process for generating random bits from the entropy source are specified in ISO/IEC 20543.

6.2.3 Additional inputs

6.2.3.1 Additional inputs overview

Additional inputs can be used to personalize the output of an RBG based on parameters such as the time/date, the intended use of the data, and other user-supplied information or data, or both, that is necessary to control the internal functionality of the RBG. These inputs typically include either commands or time-variant parameters, or both, such as a clock or internal counter. The additional input may be used to insert new entropy into the RBG. The entropy of the output shall not depend solely upon any properties of the additional inputs.

Additional input can be inserted during instantiation, generation, and reseeding.

6.2.3.2 A requirement for additional inputs

The form and use of additional inputs shall not degrade the entropy of the RBG.

6.2.4 Internal state

6.2.4.1 Internal state overview

The internal state contains the memory of the RBG and includes both secret and non-secret information. It can include a seed, a counter, a clock value, user input, the security strength of the generation process, the security strength of output, etc. The internal state consists of all the parameters, variables, and other stored values that the deterministic parts of the RBG use or act upon. The internal state is usually implicit in the specification of the deterministic functions for which it exists; the model makes the state explicit in order to better address the security issues that it can affect.

Functionally, the internal state plays two different roles. The first is to ensure that the output appears random even when the entropy of the input is insufficient to generate a truly random sequence of that length. This randomness property is ensured by the output generation function. Since the internal state transition functions and the output generation function are deterministic functions, and since deterministic functions have the property that they always give the same output for the same input, it can be necessary to have a state variable that is constructed to change with each block of RBG output. The purpose of this variable is to ensure that the output appears random even though no, or little, true entropy has been added to the system (e.g. since it was reseeded with fresh entropy).

This is particularly relevant for a DRBG. For a DRBG, unless the seed or secret parameter is updated by an external device, it is incumbent upon the design that only in the remotest possibility can a repeat sequence of the state occur. Therefore, the seed (or the secret parameter) shall be updated periodically or the device shall become inoperable after a predetermined period of time.

The second role of the internal state is to parameterize either the deterministic internal state transition functions or the output generation function, or both, in such a way that when this parameter is unknown, the deterministic function exhibits the properties required of it. For example, it can parameterize the output generation function in such a way that the internal state cannot be deduced from the output when the secret parameter is unknown. The secret parameter is typically one or more cryptographic keys, and these shall be generated in a suitably random fashion. When such a secret parameter exists, it shall have a well-defined security life cycle, which includes provision for its generation, distribution, update and destruction.

NOTE 1 It is optional for RBGs to have a secret parameter.

NOTE 2 Key management concerns are discussed in ISO/IEC 11770-1.

6.2.4.2 Requirements for the internal state

The requirements for the internal state of an RBG are as follows.

1) The internal state shall be protected in a manner that is consistent with the use and sensitivity of the output.

2) The internal state shall be functionally maintained properly across power failures, reboots, etc. or shall regain a secure condition before any output is generated (i.e. either the integrity of the internal state shall be assured, or the internal state shall be re-initialized).

3) The state elements that accumulate or carry entropy for the RBG shall have at least *x* bits of entropy, where *x* is the desired security strength expressed in bits of security. The state elements should accumulate more than the minimum number of bits of entropy for reasons of assurance and to reduce the risk of using identical values in two different cryptosystems. Generally, it is recommended to keep the entropy in the internal state somewhat larger than *x*, e.g. *x*+64 typically provides a sufficient margin, while if very large amounts of pre-computation and memory are available to an adversary, a size of 2*x* should be used (e.g. Reference [19]).

4) The secret portion of the internal state shall have a specified finite period after which the RBG shall either cease operation or be reseeded with sufficient additional entropy. Operations using an old seed shall cease after the specified period elapses unless a new seed is provided. See ISO/IEC 11770-1 for further details on the key life cycle.

5) A specific internal state shall not be reused, except strictly by chance. This implies that the same seed shall not be deliberately input to a different instance of a DRBG.

6.2.4.3 Recommendation for the internal state

The internal states that are used to produce public data, (e.g. nonces and initialization values), should be fully independent from the states used to produce secret data such as cryptographic keys.

6.2.5 Internal state transition functions

6.2.5.1 Internal state transition functions overview

Internal state transition functions alter the internal state of the RBG. In a DRBG, the internal state transition functions are composed of one or more cryptographic algorithms that are part of the generator specification. The internal state transition functions encompass everything in the RBG that can set or alter the internal state. As a matter of convention, neither the randomness source nor any other RBG data inputs are considered to act directly on the internal state. Instead, the randomness source and other RBG data inputs are arguments to functions that act on the internal state.

Since the internal state can comprise several distinct components, the internal state transition functions are likely to be composed of several distinct collections of functions, distinguished by the state component upon which they act. The functions in such a collection can be as trivial as the identity function or as complex as a cryptographic function. For a clock or counter register, the function that sequences the register as a clock or counter is a member of the internal state transition functions. While the output of these functions is always directed to a component of the internal state, the input may be from the randomness source, another external RBG input, various components of the internal state, or any combination thereof.

The internal state transition functions shall also allow for the secure manipulation (e.g. updating) of the secret parameter. Access to these functions shall be strictly controlled.

In a DRBG, the internal state transition functions are responsible for much of the security of the generator, including how much output the RBG may produce for a given randomness source input.

In an NRBG, the internal state transition functions determine how the entropy collection affects the internal state of the RBG. These functions can work in a variety of ways: they can be simple, stateless algorithms, or can combine previous states with newly collected entropy in order to accumulate entropy.

6.2.5.2 Requirements for the internal state transition functions

The requirements for the internal state transition functions of an RBG are as follows.

1) The internal state transition functions shall be verifiable via a known-answer test.

2) The internal state transition functions shall, over time, depend on all the entropy carried by the internal state.

3) The internal state transition functions shall resist observation and analysis via power consumption, timing, radiation emissions, or other side channels, as appropriate.

4) It shall not be feasible (either intentionally or unintentionally) to cause the internal state transition functions to return to a prior state in normal operation (this excludes testing and authorized verification of the RBG output).

6.2.5.3 Recommendation for the internal state transition functions

The internal state transition functions should enable the RBG to recover from the compromise of the internal state (i.e. provide enhanced forward secrecy) through a periodic incorporation of entropy.

6.2.6 Output generation function

6.2.6.1 Output generation function overview

The output generation function (OGF) acts on the internal state to produce output bits, as requested by the consuming application. A request to the OGF contains (at least implicitly) the number of output bits required and the required minimum security strength for those bits. Such a request can cause the internal state to be updated by the internal state transition functions. The OGF accepts the working state component of the internal state as input and produces the RBG output. It can be as complex as a cryptographic function. The OGF can format or block the output to conform to an external interface convention. The OGF should be a “one-way” function, so that it is hard to invert the OGF in order to recover part of the internal state.

The OGF is deterministic and always produces the same output for any particular value of the internal state. Therefore, there is an important relationship between the OGF and the internal state transition function. The internal state transition function shall always be invoked at least once to update at least the working state between successive actions of the output generation function.

6.2.6.2 Requirements for the output generation function

The requirements for the output generation function of an RBG are as follows.

1) The output generation function shall be deterministic (given all inputs) and shall be testable by a known-answer test. The known-answer test output shall be separated from operational output.

2) The output generation function shall use information from the internal state that contains sufficient entropy to support the requested security strength.

3) The output shall be inhibited until the internal state exhibits/obtains sufficient assessed entropy.

4) Once a particular internal state has been used for output, the internal state shall be changed in order to produce more output.

5) The output generation function shall resist observation and analysis via power consumption, timing, radiation emissions, or other side channels, as appropriate.

6) The output generation function shall resist attempts to interfere with its normal operation (i.e. fault attacks) as required by the operational environment of the RBG.

7) The output generation function shall protect the internal state, so that the analysis of RBG outputs does not reveal useful information about the internal state.

6.2.7 Health test

6.2.7.1 Health test overview

Health tests are concerned with assessing the health of the RBG. Health tests include functions that:

- assess the entropy of the input;
- assess the statistical quality of the output; and
- check whether the internal functions have been compromised.

The internal functionality (i.e. the "health" of the deterministic portions of the RBG) can most effectively be checked using statistical tests on the noise source and known-answer tests on deterministic components. The health of the entropy source and the quality of the output shall be checked using statistical techniques or tests.

Detailed information on assessing the health of NRBGs and DRBGs can be found in 8.8 and 9.8, respectively.

6.2.7.2 Requirements for the health test

The requirements for health tests of an RBG are as follows.

1) An RBG shall be designed to permit testing that ensures the generator operates correctly.

2) When an RBG fails a test, the RBG shall enter an error state and output an error indicator. The RBG shall not perform any random output generation while in the error state (see 8.8.4). The RGB shall be able to recover and return to the operational state (e.g. repair and test) if the permanent error state is deemed unacceptable for the application requirements.

7 Types of RBGs

7.1 Introduction to the types of RBGs

Every RBG shall have a primary randomness source. RBGs are classified into two basic types, depending on the nature of their primary randomness source. The primary randomness source is either an entropy source or a seed value.

If randomness is obtained from a system from which any amount of entropy can be extracted by sampling (provided that this system runs for a sufficiently long period of time), then it is deemed to be non-deterministic. In particular, no deterministic algorithm should be able to predict the output of a non-deterministic randomness source. The primary randomness source for an NRBG is an entropy source, which provides non-deterministic input.

If randomness is obtained from a deterministic process, then the output of that process is deemed to be deterministic. A DRBG is instantiated with a seed value obtained as specified in 9.3.2. The DRBG produces deterministic output if no non-deterministic input is provided during the generation process. An RBG may also utilize additional randomness sources that can be either deterministic or non-deterministic. The amount of entropy required by the consuming application shall be provided by an appropriate combination of these randomness sources.

Care shall be taken to ensure that an adversary cannot gain sufficient control over the randomness source in order to degrade the entropy in the RBG output.

This document is concerned with the generation of sequences of random bits. Some applications can require the generation of sequences of random numbers. Annex B specifies the conversion methods for generating sequences of random numbers from sequences of random bits which shall be used to generate sequences of random numbers.

7.2 Non-deterministic random bit generators

Non-deterministic random bit generators (NRBGs) can be further specified into sub-classes depending on the nature of their entropy source.

A physical entropy source is one in which dedicated hardware is used to measure the physical characteristics of a sequence of events in the real world. Such devices typically continue to provide an output, provided that power is applied to the measuring device. Examples of physical entropy sources and their measurement include measuring the time between radioactive emissions of an unstable atom and measuring the noise characteristics of an unstable or “noisy” diode. For more details about the requirements of physical entropy sources, see 8.3.2.

A non-physical entropy source is any entropy source that is not a physical entropy source. Examples of measuring the output of non-physical entropy sources include measuring the time between key presses or sampling data in regularly used portions of random-access memory (RAM) memory. For more details about the requirements of non-physical entropy sources, see 8.3.3.

A physical NRBG is an RBG with a physical entropy source. A non-physical NRBG is an RBG with a non-physical entropy source.

A pure NRBG obtains input only from a physical or non-physical entropy source.

Hybrid NRBGs obtain entropy from entropy sources and seed values from deterministic sources. A hybrid NRBG shall satisfy all the security criteria imposed on pure NRBGs and shall also satisfy certain extra security requirements (see 8.3.5) that are typical for DRBGs. The advantage of using a seed value from another randomness source as an additional input to the NRBG is that it allows the output of the NRBG to be parameterized by the user.

Multiple users can use the same NRBG and its associated entropy source as long as the entropy is not reused (e.g. the same entropy bit-string is not provided to more than one user or to the same user repeatedly).

NOTE 1 Additional information on classes of RBGs can be found in References [14] and [15].

NOTE 2 The use of a secret seed value, for example, can be an additional DRBG-type security feature. If the NRBG meets the requirements of this document even with a simple state transition function and a simple output function (see 8.2 and E.1), a cryptographic state transition function or a cryptographic output function can be an additional DRBG-type security feature.

7.3 Deterministic random bit generators

Deterministic random bit generators (DRBGs) can also be classified as either pure or hybrid. A DRBG is considered "pure" if the only external input is the initial seed and is considered "hybrid" if it is capable of accepting external input values during operation.

A hybrid DRBG shall satisfy all the security conditions imposed on pure DRBGs and shall also satisfy certain extra security requirements (see 9.3.4).

The security of a pure DRBG depends only on its initial seed and its cryptographic algorithm(s) for producing output. The security of a hybrid DRBG also depends on the randomness and frequency of its external input.

NOTE 1 If an RBG can be viewed as both a hybrid NRBG or alternatively as a hybrid DRBG, it is denoted as hybrid RBG. Loosely speaking, hybrid RBGs have two security anchors.

The advantage of using a hybrid DRBG is that it can help prevent either cryptanalysis or add security features such as enhanced forward secrecy, or both.

NOTE 2 Additional information on classes of DRBGs can be found in References [13] and [15].

7.4 The RBG spectrum

The classes of RBGs are depicted in Figure 2.

The distinction between an NRBG and a DRBG can be seen as a sub-division of a spectrum of design choices. At one end of the spectrum is a DRBG that is designed to use a single seed for its entire lifetime. At the other end of the spectrum is a simple NRBG that is designed as the output of a strong, highly random, non-deterministic entropy source. Intermediate RBG designs allow for periodic reseeding of DRBGs. Intermediate RBG designs allow for NRBGs whose output also depends upon a seed value or NRBGs that process the output of the entropy source in a complex manner similar to the way in which a DRBG processes a seed value. The choice of type of RBG is a cost/benefit trade-off depending on the application requirements.

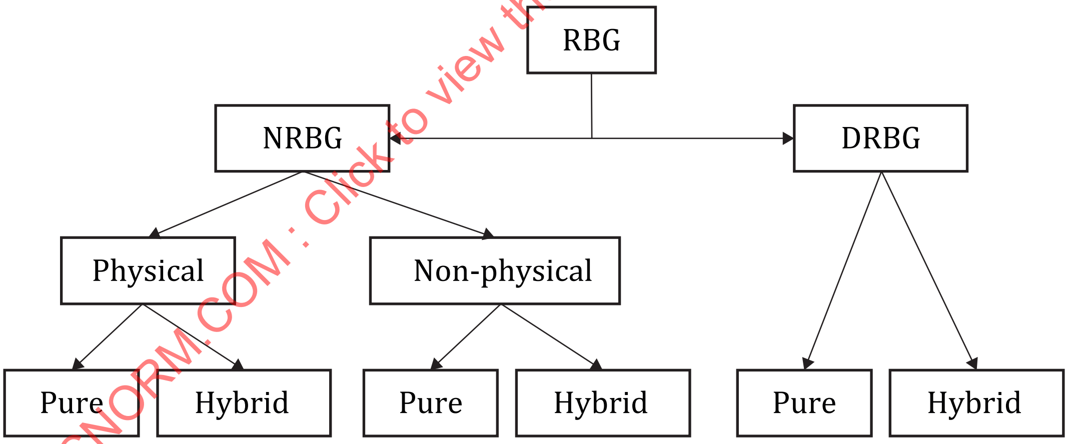

Figure 2 — RBG classes

8 Overview and requirements for an NRBG

8.1 NRBG overview

A consuming application may choose to use a DRBG for random bit generation. In such a case, the primary role of an NRBG is to generate random initial seeds for a DRBG. However, this document does not preclude using an NRBG, potentially with additional strong cryptographic post-processing, to generate all of the randomness required by an application.

Objectives and requirements that are unique to NRBGs, beyond the requirements in Clauses 5 and 6, are specified in 8.3 to 8.9. Examples of NRBGs can be found in Annex D.

8.2 Functional model of an NRBG

This subclause introduces a specialization of the components and the general model for the case of an NRBG. It describes the general operation of an NRBG, including the objectives that each NRBG functional component is intended to accomplish.

Figure 3 provides a functional block diagram depicting a conceptual NRBG. In this diagram, dashed lines indicate a component that is optional. It is important to note that the mandatory components shown are not required to be implemented as actual physical components but shall be implemented functionally.

Each of the components and their objectives and requirements are intended to prevent various security weaknesses associated with random bit generation that have been known to occur in cryptographic applications and environments. In general, each of the following components are required in an NRBG. In some applications, it can be the case that none of the requirements for a given component are applicable. If this can be adequately justified and documented, that component can be omitted from the NRBG, either because the threat being countered by the component is not present in the intended application, or because the objective served by the component is being met implicitly or addressed by other means.

The following paragraph is an overview of the way in which these components interact to produce random output. The non-deterministic entropy source, while in general not producing acceptable random output by itself, behaves probabilistically.

An internal state transition function based on one or more deterministic cryptographic functions combines a specified quantity of the entropy source data with the working state data to produce a new working state. If the bit-length of the entropy source input data used for each internal state transition is n, and the size of the working state is m bits, then the internal state transition function is a function from the space of $(n+m)$ -bit sequences to the space of m-bit sequences. The parameter sizes typically used in an NRBG will make the number of possible inputs to this function vastly larger than the number of possible outputs from the function. This results in a very large number of combinations of input sequence and current working state being assigned to any particular output for the new working state. Furthermore, if the cryptographic functions upon which typical internal state transition functions are based have been thoroughly analysed cryptographically, it is reasonable to assume that such an internal state transition function will map the input space to the output space in a nearly uniform way (i.e. each output has approximately the same number of preimages). These assumptions lead toward satisfaction of the objective of creating uniformly generated binary sequences in the working state. DRBGs and NRBGs shall add new entropy to their state at a rate greater than or equal to the rate that they output entropy. Therefore, whenever the consuming application requires random output, both DRBGs and NRBGs ensure that the internal state contains sufficient entropy that has not yet been used to produce random output, and then use the output generation function to process the current working state to produce random output.

The output generation function takes the internal state, which can be required to be kept confidential from an adversary, and produces an output that cannot be distinguished from random. The output generation function is typically also a cryptographic function or other function having similar uniform distribution characteristics. An argument similar to that for the internal state transition function leads to the conclusion that, with appropriate parameter size choices, the output generation function produces a uniformly distributed binary output.

The behaviour of physical noise sources in physical entropy sources should be described using stochastic models (see Reference [16]). Such a model allows accurate statistical testing designed to detect defects of the noise source (i.e. when the noise source produces output that is not sufficiently random). A detailed understanding of the underlying physics of the system can be used to inform the stochastic model and can enable targeted statistical testing. It is possible to choose very simple internal state transition functions and output generation functions to facilitate the determination of the stochastic behaviour of the output bits. If the NRBG is intended to be used directly for the generation of sensitive key material (in contrast to seeding a DRBG), it is recommended to use cryptographic post-processing of the noise source output.

For non-physical entropy sources, the entropy per output bit can be lower than for physical entropy sources and it can be difficult to accurately determine the moment when the entropy source fails. In such cases, more complex post-processing operations can be required to ensure that the output of the RBG is suitably random and to cope with an undetected failure of the entropy source.

A secure NRBG shall also include mechanisms designed to increase the likelihood of continued secure operation in the event of failures or compromises. Detectable failures are addressed through the inclusion of periodic and continuous health tests on the various components. Undetectable failures or compromises are addressed in two ways: the internal state shall include either a periodically changing secret parameter (e.g. a key) or a safety margin in the maintenance of entropy during NRBG operation. In the first case, the secret parameter shall parameterize the deterministic operation of the internal state transition function so that knowledge of the working state and all the inputs to the NRBG are insufficient to determine the NRBG output. If a safety margin is used, decreases in the available input entropy (due to unexpected events or statistical model inaccuracies) are less likely to result in biased random output. An objective for an NRBG will be for the NRBG to continue to operate in a manner no less secure than a DRBG in the event that the entropy source completely fails.

This objective can be met by the inclusion of a DRBG in an NRBG design, or by:

1) verifying that the health tests on the entropy source(s) can quickly detect possible weaknesses or failures in the entropy source that can unacceptably damage the quality of the random bits; and

2) initiating appropriate measures when such errors are detected.

NOTE It is usually feasible to meet these conditions for physical entropy sources since they use dedicated hardware. A generic proposal is discussed in Reference [23].

Enhanced forward and backward secrecy are properties of a properly implemented and working NRBG, given that the inputs are unknown. In fact, it is an inherent feature of an NRBG, since the NRBG always draws upon fresh entropy for each call. Thus, enhanced forward and backward secrecy are automatically provided if the entropy source(s) deliver sufficient entropy.

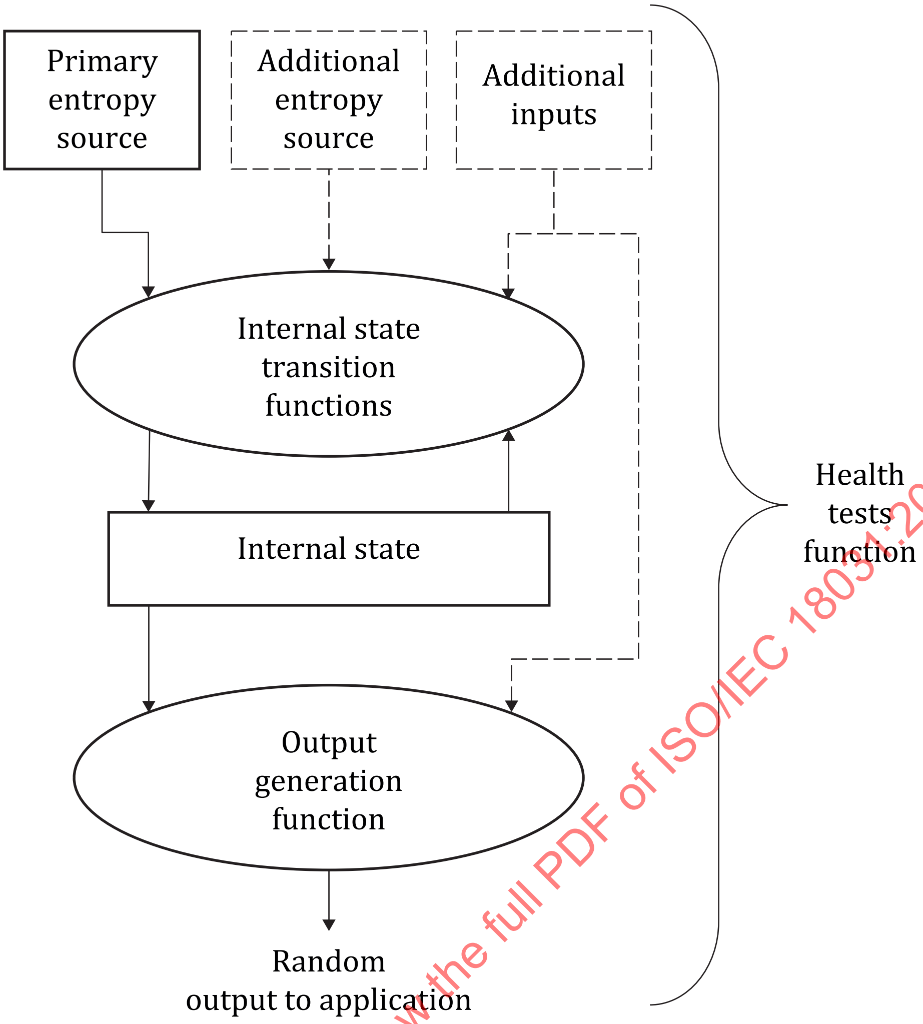
Primary entropy source
Additional entropy source
Additional inputs
Internal state transition functions
Internal state
Output generation function
Random output to application
Health tests function

Figure 3 — Block diagram of an NRBG

8.3 NRBG entropy sources

8.3.1 General

The estimation of the quantity of entropy produced by an entropy source is discussed in Annex F.

8.3.2 Primary entropy source for an NRBG

8.3.2.1 NRBG primary entropy source overview

This component serves as the source of unpredictability in the NRBG by providing data to be processed by the internal state transition function. This source of unpredictability differs from that in a DRBG, which relies on the unknown initial seed for unpredictability. In an NRBG, unpredictability is based on the use of one or more noise sources.

A physical noise source is accessed by dedicated hardware that measures the physical characteristics of a sequence of events produced by that noise source. Examples of physical noise source measurements include measuring the time between radioactive emissions of an unstable substance and measuring the output of a noisy diode that receives a constant input voltage level and outputs a continuous, normally distributed analogue voltage level.

A non-physical noise source is any non-deterministic noise source that is not a physical noise source. Examples of non-physical noise sources are system data or the RAM contents of a PC, or an aperiodic signal based on an irregularly occurring event or process timing interactions, such as the sampling of a high-speed counter whenever a human operator presses a key on a keyboard.

Specific details and requirements for physical entropy sources and non-physical entropy sources (which contain physical and non-physical noise sources) can be found in 8.3.3 and 8.3.4, respectively.

Depending on the noise source, it is possible that the output of the entropy source is not adequate for direct use as an RBG output, either because it is not in the form of a binary digital sequence or because it exhibits statistical biases. These shortcomings are remedied in an entropy source by digitization and/or conditioning the noise source output to reduce bias or ensure that the entropy rate of the entropy source output is no less than some specified amount.

The output entropy from the NRBG cannot ever be greater than the input entropy provided by the entropy source.

8.3.2.2 Requirements for an NRBG primary entropy source

The primary entropy source is the foundation of the non-deterministic behaviour of the NRBG. By definition, an NRBG design shall include this component.

The functional requirements for the primary entropy source are as follows.

1) Although the entropy source is not required to produce unbiased and independent outputs, it shall have the property that it exhibits probabilistic behaviour; i.e. the output shall not be predictable by any known algorithmic rule.

2) The entropy source shall contain a means by which the rate of entropy contribution from its constituent noise source can be assessed. This requires that the operation of the noise source be based on well-established principles or extensively characterized behaviour so that an appropriate statistical model for the noise source can be identified.

3) The noise source shall be amenable to bench testing by a validation laboratory or some other independent verification process to ensure proper operation. In particular, it shall be possible to collect a data sample from the entropy source during the validation or independent verification process to allow for the evaluation of the claimed statistical model, the entropy rate, and the appropriateness of the health tests in the entropy source.

4) Failure or severe degradation of the entropy source shall be detectable. Based on the frequency of health tests being performed, it is possible that this detection is not immediate.

5) The entropy source shall be protected from adversarial knowledge or influence. In particular, it shall be infeasible for the adversary to influence the entropy source in such a way that the entropy that the source produced falls below a threshold value without being detected.

8.3.2.3 Recommendations for an NRBG primary entropy source

The recommended features of the primary entropy sources are as follows.

1) The entropy source should be stationary in a mathematical sense. If it is not stationary, then there should be bounds on the probability of fluctuation. The probabilistic behaviour of such a source should not change significantly over time. For example, if the entropy source produces outputs from a certain alphabet with a statistical distribution, it should be consistent in this bias over time. An entropy source that is not stationary would greatly complicate the process of estimating the rate of entropy contribution and increase the difficulty of validating the resulting NRBG design.

2) Appropriate health tests tailored to the known statistical model of the source should place special emphasis on the detection of misbehavior having increased likelihood near the boundary between the nominal operating environment and abnormal conditions. This requires a thorough understanding of the operation of the entropy source.

8.3.3 Physical entropy sources for an NRBG

8.3.3.1 NRBG physical entropy source overview

The physical noise source within a physical entropy source typically provides a continuous stream of output while it has power applied. However, this output is not necessarily provided as a simple binary string. For example, if the physical noise source output is based on the time between radioactive emissions, it is possible that the noise source does not provide a simple binary output string that contains entropy but a series of timings between emissions. Alternatively, if the physical noise source used is based on voltage variations in a noisy diode, then the output can be a series of voltage measurements.

NOTE In these examples, the entropy is based on the variability of the timings (in the first example) and on assignments made for the threshold values (in the second example).

Typically, this noise source output shall be interpreted before it can be of any use. This can be done by comparing the physical data provided by the noise source to a series of threshold values. The data provided by the noise source can either be compared to a single threshold value, so that a single reading of the source produces a single bit of output, or a series of threshold values, so that a single reading of the source produces several output bits.

Since a physical entropy source is likely to be contained within a cryptographic boundary of an NRBG, the focus of the security requirements in this document is on the proper collection and interpretation of a true physical noise source. Other measures can be considered to reduce the possibility that an external adversary obtains information from, or exerts undue influence over, the noise source.

An advantage of a physical noise source is that it can generally be modeled as a stochastic process. This in turn, leads to efficient online health checks that accurately determine the moment when a physical noise source fails. Such tests can be used to prevent an NRBG based on the use of a physical entropy source from continuing to operate with a failed noise source (e.g. the noise source becomes predictable).

An evaluation methodology for physical noise sources is specified in ISO/IEC 20543.

8.3.3.2 Requirements for the physical noise source of an NRBG

The functional requirements of a physical noise source are as follows.

1) The threshold values for the noise source shall be chosen so that the output string contains a sufficient amount of entropy.

NOTE There is a difference between the entropy displayed by a noise source (which can produce data at an arbitrary degree of precision) and the entropy provided by a binary interpretation of that source.

2) The total failure of the noise source shall be immediately detectable. A degradation of the noise source shall be detected quickly.

8.3.3.3 Recommendation for the physical entropy source of an NRBG

The entropy source should be formally analysed and the threshold values for its noise source chosen in such a way that the output contains the greatest possible amount of entropy. Making such an analysis is feasible because true physical noise sources are comparatively simple compared to entropy derived from non-physical sources.

8.3.4 NRBG non-physical entropy sources

The entropy from non-physical noise sources within non-physical entropy sources is usually provided by a system upon request from the NRBG. Hence, non-physical noise sources are typically outside the defined protection boundary of the NRBG. The data is usually already binary in nature. For example, the noise source output can be a digital representation of the time between key presses (as measured by a system), certain unpredictable network statistics, or the contents of unpredictable portions of the RAM.

Since a non-physical entropy source is, at least partly, outside the control of the NRBG, sufficient precautions shall be taken to minimize the possibility of an adversary gaining any knowledge of the data and/or the likelihood of influencing the entropy source. Furthermore, as it is a lot harder to model a non-physical noise source accurately as a stochastic process, it can be more difficult to determine whether such a source is operating correctly or not. Hence, NRBGs based on non-physical entropy sources often use complex state transition and output generation functions that guarantee that the RBG is at least as secure as a DRBG for the period of time between the noise source failing and the failure being detected.

There are no extra security requirements for a non-physical entropy source.

An evaluation methodology for non-physical noise sources is specified in ISO/IEC 20543.

8.3.5 NRBG additional entropy sources

8.3.5.1 NRBG additional entropy sources overview

The operation of an NRBG can also include one or more additional entropy sources. An additional entropy source can be useful for a variety of reasons. It can provide a layer of protection against degradation of the NRBG output due to the primary entropy source failing or straying from the characterized statistical model of its noise source.

In situations where the primary entropy source has some degree of external visibility, an additional entropy source that is less externally accessible lessens the usefulness of knowledge of the primary entropy source to the adversary.

Finally, having multiple entropy sources can provide the capability for split control, enabling applications where multiple users require access to the same NRBG output but distrust each other's potential influence over the individual entropy sources. For such applications, it is possible to design the NRBG so that a user's trust in a single entropy source is sufficient for trust in the final NRBG output.

An NRBG may use an additional source of randomness e.g. a seed value produced by a pure DRBG (see 9.3.2). NRBGs that use a seed value from a deterministic process are discussed in 8.3.5.2 and 8.3.5.3.

Despite the additional sources of randomness providing additional unpredictability to the output of an RBG, the security of the RBG shall rest solely upon the primary entropy source.

8.3.5.2 Requirements for NRBG additional entropy sources

The functional requirements for additional entropy sources are as follows.

An additional entropy source shall be included if the:

1) primary entropy source is insufficiently reliable from a failure perspective. In this case, the additional entropy source shall meet or exceed the same requirements as the primary entropy source;

2) primary entropy source produces entropy at a rate that is insufficient for the desired rate of random bit generation. In this case, the additional entropy source shall meet or exceed the same requirements as the primary entropy source. Alternatively, instead of including an additional entropy source, it can be acceptable to solve this problem by using the NRBG only for the initial seed generation for a DRBG; and

3) application or environment require that the RBG exhibit features that are best provided by a hybrid NRBG, i.e. that the RBG takes a seed value from another randomness source as input. In this case, the additional entropy source shall satisfy the conditions given in 8.3.6.

Whether the additional entropy source(s) is/are of the same type as the primary entropy source (i.e. a second version of the same type of entropy source), or a completely different component or process, this source shall operate independently of the primary source to ensure that the combined entropy sources will not lose entropy due to statistical dependence. Independence of entropy sources also facilitate the design and evaluation or independent verification processes by allowing the primary and secondary entropy sources to be analysed separately. It also reduces the likelihood of a failure in the primary entropy source.

8.3.5.3 Recommendations for NRBG additional entropy sources

The optional but recommended features of the additional entropy sources are as follows.

1) The RBG output should remain unpredictable even if the attacker can adaptively choose the values of the additional entropy sources; the attacker should be unable to predict the next bit produced by the RBG with probability significantly greater than one half.

2) Additional entropy source(s) should be included in the NRBG design if either of the following is true:

   a) the primary entropy source is somewhat non-stationary (i.e. inconsistent) in its statistical behaviour, making the estimation of input entropy more difficult; or

   b) there is a concern that the primary entropy source is possibly not free of adversary knowledge or influence. In this case, the additional entropy source should satisfy the same requirements as the primary source, although it can be acceptable for it to be somewhat more deterministic. That is, actions by the user or factors from the system environment can influence (although not completely determine) the output from this source in a perceptible way.

8.3.6 Hybrid NRBGs

An NRBG is a hybrid NRBG if it uses complex, stateful post-processing (e.g. cryptographic processing). The main advantage of a hybrid NRBG is that the post-processing adds computational security to the already unpredictable entropy source output as a second security anchor.

The additional functional requirements of a hybrid NRBG are as follows:

1) The post-processing shall provide backward secrecy, forward secrecy and enhanced backward secrecy.

2) The hybrid NRBG shall provide enhanced forward secrecy.

3) No unauthorized person shall be able to manipulate, influence, or update any seed value used for post-processing.

8.4 NRBG additional inputs

8.4.1 NRBG additional inputs overview

The operation of an NRBG can require taking certain public or user-generated inputs such as commands, power variations and time-variant data such as counters, clocks or user-supplied data. It may be assumed that these additional inputs are either directly observable or under the direct control of an adversary. Therefore, it is vital that the manipulation of these inputs does not reduce the effectiveness of the RBG or that only correctly authenticated and authorized personnel have the ability to manipulate these inputs, and then only within a defined operational policy.

8.4.2 Requirements for NRBG additional inputs

The functional requirements of all additional inputs shall include protection against their manipulation (commands, clock, timers, power etc.) that comprises the security of the RBG.

NOTE This can be accomplished by limiting the influence that these inputs have over the overall control of the NRBG. Power input is, of course, a special case. Disruption of power will obviously result in a complete denial of service. If this is a concern, then the operating environment of the NRBG is expected to provide uninterruptible power, which is a system issue and beyond the scope of this document.

8.5 NRBG internal state

8.5.1 NRBG internal state overview

This component consists of information that is carried over between calls to the NRBG and all the information that is processed during a request. For this reason, an internal state is a requirement; however, it is not compulsory that any portion of the internal state depend upon previous states, i.e. there is no requirement for any portion of the internal state to be carried over to the next NRBG call (e.g. in the coin flip example, no internal state is carried over from one coin flip experiment to the next). In such cases, the internal state of an NRBG is totally dependent on the output of the entropy source at the time that the NRBG is used.

However, by retaining this state information, the NRBG can produce a random output as a function of not only the current input from the entropy source but also several (or all) previous inputs. This provides a layer of protection against entropy source failure or degradation, as well as a compromise of the random output by an adversary who has knowledge of or influence on the entropy source.

The internal state consists of two parts. The working state is the portion of the internal state that is processed in combination with entropy source data by the internal state transition function to produce the new internal state. This portion can consist of a random “pool,” in addition to any optional counters or other values. The second part of the internal state is a value referred to as a secret parameter. The secret parameter is an additional input to the internal state transition function that customizes the function for that particular instance of the NRBG. As such, it serves as an additional layer of protection against a degraded or compromised entropy source. Depending on the handling of the secret parameter, it can also protect against a compromised working state.

Since the entropy source output shall initially be placed into an internal state, the working state is mandatory. All, part, or none of the internal state may be carried forward to the next NRBG invocation. The use of a secret parameter is optional but where used, it is typically preserved between NRBG invocations and can only be updated by an authenticated and authorized person (or personnel) within the boundary of the operational policy. More information on the issues associated with control of the secret parameter is provided in ISO/IEC 11770-1.

8.5.2 Requirements for the NRBG internal state

The functional requirements of the internal state are as follows.

1) The design of the NRBG shall protect against knowledge of or influence over the internal state by an adversary.

NOTE 1 Possible means of accomplishing the above requirement include assigning the internal state to a memory region accessible only to the NRBG, hosting the NRBG on a standalone computer or device, or through security policies that physically protect the system and its environment.

2) The secret parameter, if it exists, shall be protected from adversarial knowledge by the NRBG protection boundary, which shall be designed to detect unauthorized penetration attempts.

3) The initial value of the secret parameter, if it exists, shall contain sufficient entropy to meet the security requirement of the NRBG. This initial secret parameter shall either be generated by the NRBG or by another NRBG. If the secret parameter is generated by the NRBG itself, the NRBG shall operate in a special mode dedicated for this purpose, where the resulting random output becomes the secret parameter.

   However, if additional security is deemed to be a requirement to protect against possible scenarios of the adversary gaining knowledge of or influencing the entropy source, then the initial secret parameter generation process shall obtain additional entropy data either from another system component or through interaction with the user for combining in some way with the entropy source data (i.e. this additional entropy data serves as a temporary secondary seed value).

NOTE 2 Examples of user interaction can consist of, but are not limited to: key presses, timings between key presses, or mouse movements.

4) In the event that there are values that the secret parameter should not take (e.g. “weak” cryptographic keys), then the secret parameter shall be tested to make sure that these secret parameter values are not used.

5) The secret parameter, if it exists, shall be replaced periodically. This supports the objective of enhanced forward and backward secrecy. The secret parameter should only be updated through commands issued by properly authenticated and authorized personnel within the boundary of the operational security policy or by automated methods in accordance with established criteria (e.g. based on time or the number of generation requests).

6) If a secret parameter exists and unless it is obtained from another NRBG, the secret parameter replacement or updating process shall involve the entropy sources and internal state transition function.

NOTE 3 A reasonable replacement scheme would be simply to replace or add the current secret parameter with ordinary random output from the NRBG that has not and will not be used for any other purpose.

7) If the entropy source(s) fails and the working state becomes compromised, the NRBG shall resist any attempts to force it to produce predictable output.

NOTE 4 This can be achieved by ensuring that either the NRBG ceases operation when the entropy source(s) fails, or the NRBG includes a secret parameter. However, both can only be applied if the individual use case or statutory stipulations permit the NRBG to continue to operate in a manner that is no less secure than a DRBG, in particular if the secret parameter is of sufficient length and strength to resist any form of cryptanalytic attack against the NRBG, including exhaustive search.

8.5.3 Additional information for the NRBG internal state

The optional but recommended features of the internal state are as follows.

1) The size of the internal state, in bits, should be sufficient to enable the NRBG to continue to act as a DRBG if the entropy source(s) fails or becomes completely known to, or controlled, by an adversary. If the entropy source data becomes constant, the maximum possible cycle length is bounded by the size of the internal state, and this places an upper bound on the work that an adversary would need to perform to recover the internal state (through exhaustive search).

2) The secret parameter should be preserved between operational sessions in order to provide the NRBG with a unique state with sufficient entropy at each power-up initialization without having to immediately create a new secret parameter value. If this is done, the parameter should be preserved in a way that protects it from adversarial access.

NOTE This protection can take the form of storage in a memory area accessible only to the NRBG process, storage in encrypted form, or storage in a removable token that is subsequently protected (e.g. by storing in a vault).

8.6 NRBG internal state transition functions

8.6.1 NRBG internal state transition functions overview

The internal state transition functions control all the operations that alter the internal state. These include the mandatory functions that place the output of the entropy source into the working state and that present part of the internal state to the output generation function. Typically, these functions act independently of the secret parameter.

However, the main function of the internal state transition functions is to control the “carry over” parts of the internal state between invocations of the NRBG. This functionality is optional but recommended. Such a function typically works in two parts.

1) It carries the secret parameter over to the subsequent internal state without change.

2) It updates the working state using a function that depends upon the current working state and (optionally) the secret parameter.

Figure 4 shows an example of an internal state transition function with a secret parameter.

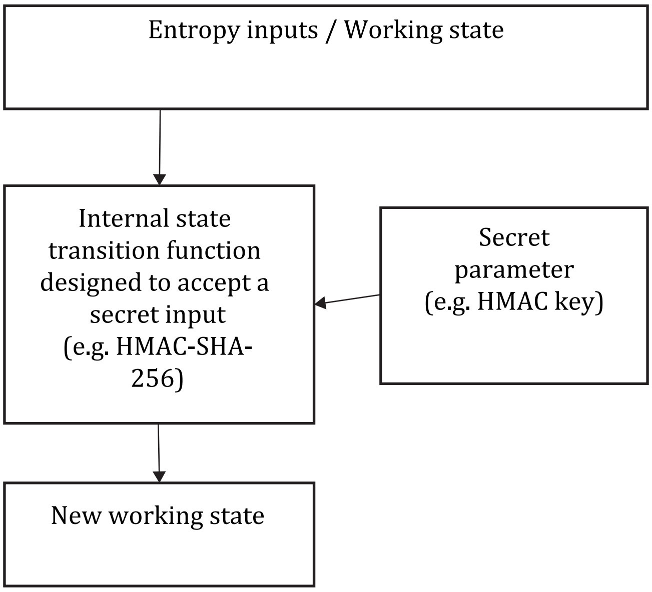
Entropy inputs / Working state
Internal state transition function designed to accept a secret input (e.g. HMAC-SHA-256)
Secret parameter (e.g. HMAC key)
New working state

Figure 4 — Example of the internal state transition function

8.6.2 Requirements for the NRBG internal state transition functions

The use of an internal state transition function is required only to collect and store the output of the entropy source(s). If this is the only function that the internal state transition functions provide, then no additional requirements apply to the internal state transition functions.

If the internal state transition functions produce a new working state by combining the previous internal state with the output of the entropy source, then the functional requirement of the internal state transition function is as follows:

— For each replacement of the entropy-accumulating portion of the internal state, the entropy source data processed by the internal state transition function shall be of sufficient quantity to contain at least as many bits of entropy as the security strength of the NRBG.

8.6.3 Recommendations for the NRBG internal state transition functions

The optional but recommended features of the internal state transition functions are as follows.

1) The internal state transition functions should achieve backward secrecy through appropriate use of a one-way function, such as a cryptographic hash-function.

2) The internal state transition functions should have the property that all bits in the working state and the entropy source input should influence the internal state transition function's output bits (i.e. the content of the updated internal state).

3) The operation of the internal state transition functions should be protected against observation and analysis via power consumption, timing, radiation emission, or other side channels, as appropriate. The values that the internal state transition functions operate on (e.g. the internal state including any secret parameter and entropy source input) are the critical values upon which the confidentiality of random output is based. Side channel analysis can potentially defeat this confidentiality.

8.7 NRBG output generation function

8.7.1 NRBG output generation function overview

This component provides random output to the requesting application by processing all or a subset of the bits in the current internal state (both the working state and, possibly, the secret parameter) and possibly any optional additional inputs. The output generation function serves as an important component in obtaining backward and forward secrecy. The component provides backward secrecy by preventing the random output from revealing information about the previous or current values of the internal state, entropy source inputs, or random outputs.

The output generation function may provide:

- forward secrecy by preventing output values from revealing internal state information; or
- enhanced forward secrecy by triggering the insertion of fresh entropy;
- or both.

8.7.2 Requirements for the NRBG output generation function

The functional requirements for the output generation function are as follows.

1) The output generation function shall not introduce biases into the random output.

2) The random output from the output generation function shall pass the statistical health tests when they are required (see 8.8.5).

3) The output generation function shall process at least as many bits from the internal state as the number of bits in each random output block produced by the output generation function. Depending on the type of output generation function used, it can be necessary to process significantly more than this number of bits from the internal state.

4) The output generation function shall not reuse data from the working state portion of the internal state when providing random data to the requesting application. That is, either:

    a) the internal state transition function shall replace sufficient portions of the working state between calls to the output generation function to ensure no reuse; or

    b) the output generation function shall use a previously unused portion of the working state to ensure that data is not reused.

5) The output generation function shall not leak any information about the internal state that may potentially enable future output to be compromised. That is, the output generation function shall not be able to be inverted to reveal information about the internal state, and its output shall not reveal any information about the current state.

8.8 NRBG health tests

8.8.1 NRBG health tests overview

This component ensures that the overall NRBG process continues to operate correctly and that the NRBG output continues to be random. These tests should detect failures in the NRBG and prevent the NRBG from being used until the health tests can be successfully passed. These tests are an integral part of the NRBG design. They are performed automatically at power-up or initialization without intervention by other applications, processes or users, and can also be requested by the user at any time.

The health tests presented in this subclause include three sets of tests:

- tests on the deterministic components of the NRBG;
- tests within the entropy sources; and
- tests on the random output produced by the NRBG.

The tests on the deterministic components shall apply to all NRBG designs. There can be cases where the rate of entropy acquisition or random output is too low to feasibly implement all statistical health tests on either the entropy sources or the NRBG output. In such cases, the designer may attempt to modify the tests or test thresholds, to permit smaller sample sizes, while keeping the Type 1 error probability approximately the same.

NOTE For the purposes of this document, a Type 1 error is defined in terms of a test of a hypothesis in which an assessment of a random sequence results in a decision that the sequence is not random when in fact it is random. Additional information about Type 1 errors can be found in Reference [17].

General requirements and specific requirements to each of these sets of tests are presented in 8.8.2 to 8.8.5. In some cases, the NRBG can have additional features or functionality not addressed by this document and can include specialized tests. In these cases, the designer shall thoroughly document the objectives of these additional features and the basis for the additional tests.

The frequency at which health tests shall be performed depends on the overall design of the RBG. For example, if it can be ensured that any failure of the entropy source can be detected quickly, as assured by frequent testing of the entropy source, the deterministic component can be simplified.

8.8.2 General NRBG health test requirements

The functional requirements for all three categories of health tests introduced in 8.8.1 are as follows.

1) The NRBG shall automatically perform thorough health tests at each power-up or initialization.

2) The NRBG shall allow for health tests (on the entropy sources, deterministic components, and random output) to be performed “on demand.”

3) All outputs from the NRBG shall be inhibited while health tests are being performed in order to conceal information about the operation of the NRBG and to prevent the release of any information about possible failures (this includes the health tests on deterministic components in 8.8.3, health tests on entropy sources in 8.8.4 and health tests on random output in 8.8.5. Data that has successfully passed all tests on random output can be used as random output following the completion of all health tests.

NOTE 1 An exception to this requirement can exist if health tests are continually applied to the NRBG (e.g. health tests within the entropy source or statistical tests on the noise source output). In this scenario, it would be impractical if not impossible to meet this requirement.

4) If the NRBG is implemented as software or firmware, the health tests performed at initialization shall include an integrity check on the implementation code (e.g. RAM, Read-Only Memory (ROM) or programmable logic device). Examples of ways to do this include a digital signature or message authentication code applied to the software or firmware.

If increased assurance is desired, the implementation of the NRBG requires the performance of the health tests with increased frequency during an operational session without serious degradation in performance in addition to during power-up initialization (a reasonable interval is one based on the frequency of requests for random bits).

5) If any of the health tests return a failure result, the NRBG shall enter an error state and indicate a failure condition to the application or user. The NRBG shall not perform any random output generation while in the error state. The NRBG shall require user intervention (e.g. power cycling or re-initialization), followed by successful passing of the health tests, in order to exit the error state.

NOTE 2 For example, to recover or exit from an error state, an NRBG is required to follow its maintenance procedures.

8.8.3 NRBG health test on deterministic components

The objective of these tests is to ensure that the deterministic components of the NRBG continue to correctly process any possible set of inputs. Since, by definition, there is no unpredictability in these components, the accepted method of testing is to use known-answer tests. Such tests initialize the component or function to a fixed initial state, input a fixed input to the function, then compare the resulting output with the correct output that was computed previously by an independent implementation of the function (e.g. a verified computer simulation used during NRBG development) and stored with the NRBG implementation.

The functional requirements for the health tests on the deterministic components are as follows.

1) The known-answer tests shall be included in the overall health tests performed either at each power-up or re-initialization, or both, and on demand. The known-answer tests should be included in the overall health tests performed at periodic intervals.

2) The comparison sequence (the result to be compared with the known answer) produced for any known-answer test shall be sufficiently long so that the probability of passing the test with failed or degraded components is acceptably low. Since 32-bit checksums are used in many information assurance applications, 32 bits is a recommended minimum length for known-answer test values.

3) The known-answer tests shall be performed on all deterministic components of the NRBG, including not only each component of the NRBG, but also the overall NRBG control components. This can be done by setting the internal state to a fixed pattern, blocking the entropy source data and replacing the entropy source output with a fixed bit sequence, and running the NRBG process in its operational mode. This action shall produce an output sequence of at least the length determined according to requirement 2) above. The output is then compared to the predetermined value developed through independent implementation or simulation.

4) The known-answer tests shall exercise each aspect of the function being tested. This generally requires that the fixed input pattern be long enough to provide a representative sample of possible inputs to each major functional component of the function being tested.

5) All output from the NRBG shall be inhibited while known-answer health tests are being performed in order to conceal information about the operation of the NRBG and to prevent the release of any information about possible failures. Data resulting from the known-answer health tests shall not be used as random output following the completion of all health tests (since the data is to be used only for health testing). The current value of the NRBG internal state following successful completion of the known-answer test shall not be used except for comparison with the expected internal state.

The known-answer tests on the deterministic components of the NRBG may be eliminated in favour of implementing the NRBG as two redundant and independent processes (other than the entropy sources) whose outputs are continuously compared. In this case, a mismatch shall be handled as a health test failure, with entry to the error state.

8.8.4 NRBG health tests within entropy sources

The objective of these tests is to detect variation in the behaviour of the noise source from the intended behaviour. It is possible that the noise source output is biased and not independent. The entropy can also be very sparse. In the vast majority of cases, traditional randomizer tests (e.g. monobit, chi square and runs tests that test the hypothesis of unbiased, independent bits) virtually always fail and thus are not useful. In general, tests within the entropy sources are required to be tailored carefully to the noise source, taking into account the non-uniform statistical behaviour of the correctly operating noise source.

For non-deterministic noise sources obeying very complicated statistical models, and for non-physical noise sources in particular, it is possible that it is not feasible to develop statistical tests that correspond precisely to a statistical model of the noise source. In these cases, it can be more appropriate to identify simpler tests that, instead of determining whether a statistic calculated from a data sample falls within an acceptable range, determine instead whether the data sample contains any occurrences of values known to be associated with failures. The selection of the patterns used for such tests should take into account the noise source's likely failure behaviour.

The functional requirements for the health tests within the entropy sources are as follows.

1) The tests on the noise sources shall be included in the overall health tests performed either at each power-up or re-initialization, or both, at periodic intervals during use, and on demand.

NOTE 1 For example, the periodic interval for tests on the noise source can be determined by the module's implementation policy. The noise source is continuously tested to quickly detect catastrophic failures that cause the entropy noise to become stuck on a single output value and detect a large loss of entropy that can occur as a result of some physical failure or environmental change affecting the noise source.

2) As a minimum, each noise source shall be tested for activity. That is, the test shall collect a quantity of data from the source and confirm that it does not consist solely of a constant output or other repeated pattern.

   NOTE 2 Constant outputs are those consisting only of a single value of the digitized noise source output. For example, if the noisy diode in the example produced the value 0110 at each sampling, it would fail an activity test.

   The size of the data sample collected depends on the characteristics of the noise source and shall be chosen such that when the noise source is operating correctly, the probability of no activity within a sample of that size is acceptably low (10^{-4} is a recommended value for this Type 1 error rate).

3) The tests applied to each of the noise sources and rationale for their appropriateness shall be thoroughly documented. The rationale shall indicate why the tests are believed to be very likely to detect failures in the noise sources.

The tests on each entropy source shall include tests that take into account the known characteristics of the noise source. An evaluation methodology for health tests is specified in ISO/IEC 20543.

8.8.5 NRBG health tests on random output

The objective of these tests is to provide a final check on the randomness of the output from the NRBG. The deterministic tests on the internal state transition function and output generation function are easier to conduct than testing the non-deterministic outputs of the entropy source. These functions typically do such thorough mixing that even a complete failure of the entropy sources would not cause detectable statistical irregularities in the random NRBG output. This is a consequence of the objective that the NRBG continue to operate in a manner no less secure than a DRBG if the entropy sources fail or come under the influence of an adversary. However, statistical tests on random output are still useful, and are addressed in the requirements below.

The functional requirements for the health tests on the random output are as follows.

NOTE 1 The functional requirements mainly address statistical tests. As indicated above, statistical tests are often ineffective in validating the quality of the random output when strong cryptographic post-processing is applied.

1) The tests on the random output shall be included in the overall health tests performed at each power-up or re-initialization, or both, at periodic intervals during use, and when requested by the user.

2) The NRBG shall be tested for failure to a constant value by performing a test on each block of random output produced by the output generation function. See 9.8.8 for continuous RBG test specifics.

As a minimum, the NRBG health test should include the following set of tests on a sequence of 20 000 bits of random output from the NRBG.

The overall set of tests on the NRBG shall be deemed to have passed if all four individual tests are passed. The indicated thresholds below correspond to a Type 1 error probability of 10^{-4}.

NOTE 2 These four tests [a) to d) below] are rooted in FIPS 140-2.[18]

a) Monobit test: Let X be the number of ones in the sample. The test is passed if $9275 < X < 10275$.

b) Poker test: Divide the sequence into 5 000 consecutive four-bit segments. Count the number of occurrences of each of the sixteen possible four-bit values. Let $f(i)$ be the number of occurrences of the four-bit value i (where $0 \leq i \leq 15$). Evaluate the following: $X = \frac{16}{5000}\sum_{i=0}^{15} f(i)^2 - 5000$. The test is passed if $2{,}16 \leq X \leq 46{,}17$.

c) Runs test: A run is defined as a maximal sequence of consecutive bits of either all ones or all zeroes. The occurrences of runs for both consecutive zeroes and consecutive ones of all lengths from one to six should be counted and stored. The test is passed if these counts are each within the corresponding interval specified in Table 1 below. This shall hold for both the zeroes and ones. For the purposes of this test, runs of length greater than six are considered to be of length six.

d) Long runs test: A long run is defined to be a run of length 27 or more, composed of either zeroes or ones. The test is passed if there are no long runs.

NOTE 3 The rationale for the runs and long runs tests is discussed in Annex I.

Table 1 — Runs test intervals

Run length (bits)	Required interval
1	2 315 – 2 685
2	1 114 – 1 386
3	527 – 723
4	240 – 384
5	103 – 209
6+	103 – 209

Additional tests can be performed on the random output if:

- it is deemed that greater assurance is required. In such a case the tests a) to d) above should be augmented or replaced by more comprehensive tests (sample sizes or additional statistical tests having a Type 1 error probability of 10^{-4} or less); and/or
- a health test on random output returns a failure result; the NRBG should repeat the test a moderate number of times but shall not exceed a total of three such tests. If the random output passes the test during this set of attempts, the NRBG can resume normal operation.

The rationale for the design of statistical tests is given in Annex I.

8.9 NRBG component interaction

8.9.1 NRBG component interaction overview

The components of an NRBG are designed to accomplish certain security objectives by satisfying the specific objectives and requirements. 8.9.2 presents additional requirements involving interrelationships among the components.

8.9.2 Requirements for NRBG component interaction

1) The interaction and timing of the internal state transition function and the output generation function shall ensure that the rate at which the internal state transition function processes additional entropy source input is sufficient to provide usable data for use by the output generation function. This includes the additional constraint that the output generation function shall not reuse data from the working state. This does not necessarily require the entropy source and internal state transition function to be operating when the output generation function produces output. For example, the working state can be much larger than its minimum acceptable length (e.g. the random output block length), providing a sufficient number of bits for the production of many random output blocks.

2) The processing of entropy source input and internal state (both working state and secret parameter) by the internal state transition function shall be such that the entropy source input, working state, and secret parameter each independently contribute information into the working state after each application of the internal state transition function, regardless of the information contribution from the others. This allows independent safety margins on the maintenance of entropy within the NRBG.

8.9.3 Recommendations for NRBG component interaction

The optional but recommended features applying to the interaction of the components of an NRBG are as follows.

1) The functional components of the NRBG and the interaction among these components should be designed such that, if the entropy sources become completely degraded in a manner undetectable by the health tests, the remaining NRBG components should continue to continuously operate and interact as a DRBG. A natural way to accomplish this is by designing the NRBG as a DRBG that has been modified to operate on input from one or more entropy sources.

2) The NRBG should be designed so that the working state continues to accumulate inputs from the entropy sources even when the output generation function does not require new working state data. For example, this can be done as a background process when processor or system resources are available. This is especially recommended if there are long periods of time between application requests for random output. In these periods, the internal state can be more vulnerable to observation due to the increased length of time during which it would otherwise remain unchanged.

9 Overview and requirements for a DRBG

9.1 DRBG overview

A DRBG uses an approved deterministic algorithm to produce a sequence of bits from an initial value called a seed. Because of the deterministic nature of the process, a DRBG is considered to produce “pseudorandom” rather than random bits, i.e. the string of bits produced by a DRBG is predictable and can be reconstructed, given knowledge of the algorithm, a seed constructed from a seed value provided by a randomness source (see 9.3.2) and any other input information. However, if the input is kept secret, and the algorithm is well designed, the bit-strings appear to be random.

A DRBG is classified as either pure or hybrid, depending on its inputs. A pure DRBG only accepts a seed during initialization. A hybrid DRBG is capable of accepting input during operation (i.e. after initialization), either by reseeding with a new seed value or as additional input.

At any given time after a seed has initialized a DRBG, the DRBG exists in a state that is defined by all prior input information. The seed can be thought of as defining different instances of a DRBG, each of which outputs a sequence of output bits (possibly depending on a series of inputs, or the values of the additional randomness sources).

The security of the DRBG should depend only on the primary randomness source, although other sources can be used to provide additional randomness to maintain the unpredictability of the output even if the seed provided by the primary randomness source (the primary seed) is compromised. Hence, the primary seed shall provide sufficient randomness so that the desired security strength is achieved by the DRBG.

The security of an implementation that uses a DRBG is a system implementation issue; both the DRBG and its randomness source shall be considered.

Objectives and requirements that are unique to DRBGs, beyond those specified in Clauses 5 and 6, are specified in 9.3 to 9.9. DRBG examples can be found in Annex C.

9.2 Functional model of a DRBG

This subclause introduces a specialization of the components and the general model for a DRBG. It describes the general operation of a DRBG, including the objectives that each DRBG functional component is intended to accomplish.

Figure 5 provides a functional block diagram for a DRBG that satisfies this document. It is important to note that the components shown are not necessarily required to be implemented as actual separate program routines but are required to be implemented functionally in some sense. Each of the components and their objectives and requirements are intended to prevent various security weaknesses associated with random bit generation that have been known to occur in cryptographic applications and environments. In general, each of the following components are required in a DRBG. In some applications, there can be a legitimate argument that none of the requirements for a given component are applicable. For example, there are two common scenarios in which a DRBG can wish to update a seed: if a seed has been used for "too long" or it is suspected that an attacker has exerted some malicious control over the internal state. A re-seeding capacity is therefore not required if the device ceases operation after some limit (e.g. time, number of calls) and the internal state is suitably protected (e.g. by protective hardware).

The following is an overview of the way in which these components interact to produce pseudorandom output. The primary seed value is either directly or indirectly supplied by an entropy source.

The primary seed value can be directly supplied in one of two ways.

— The first would be if the seed value were produced by an NRBG that meets the requirements of this document (i.e. an approved NRBG).

— The second would be if the seed value were supplied by an entropy source internal to the DRBG. Such an entropy source would have to be tested to make sure that it produces output with a sufficient entropy rate in a manner similar to an NRBG.

The seed value is indirectly supplied if the seed value is produced by another DRBG meeting the requirements stated herein (i.e. an approved DRBG), which in turn shall be correctly seeded.

After the seed is initially supplied, it is loaded into the internal state by a specialized internal state transition function referred to as the seeding function. Access to this specialized internal state transition function should only be available to authorized parties. At no point shall it be possible for this seed value to be observed or altered.

After the DRBG is supplied with an initial seed and made ready for use, it does two things on demand: updates its internal state and outputs a block of random appearing data. The internal state is updated by a deterministic internal state transition function, referred to as the internal state update function in Figure 5, that takes as input the current internal state, any additional inputs supplied by the user and the outputs of any other randomness source (e.g. an entropy source or DRBG). In particular, it does not take the original seed value as input. The output of the DRBG is computed by a deterministic output generating function, which takes the current (just updated) internal state as input.

In the absence of any additional input, the initial seed determines all the outputs of the DRBG until it is reseeded. Thus, if the DRBG does not take additional inputs during use, storage of the seed in the internal state represents a major security threat; hence, if this was permitted, the DRBG could not be regarded as possessing enhanced backward security, since disclosure of the seed would threaten all security applications depending on the output of the DRBG since the seed was introduced. Therefore, as a result of requirement 11) in 5.2, a DRBG that does not take additional inputs during use shall not store the seed in its internal state.

It is often useful to think of the internal state as being the combination of a working state that is continually updated and, optionally, a series of secret parameters. These secret parameters often control the operation of the internal state transition functions and output generating function in a manner similar to cryptographic keys (and are indeed often cryptographic keys). The selection of these secret parameters is outside the scope of this document.

NOTE 1 Key management guidance can be found in ISO/IEC 11770-1.

Typically, a DRBG also contains a mechanism to update the internal state in the event that a new seed value is supplied. Such an update is performed by a specialized internal state transition function and is known as reseeding. The internal state after reseeding can depend upon the previous internal state or can involve clearing the DRBG's working state and loading the new seed in a similar manner to the initial seeding operation.

A secure DRBG also includes mechanisms designed to increase the likelihood of continued secure operation in the event of failures or compromises. The potential for failure to perform as intended is addressed through the inclusion of health tests on the various components, such as known-answer tests and continuous output tests (see 8.8.4).

Each of the DRBGs specified (see Annex C) has been designed to provide forward and backward secrecy when observed from outside the DRBG boundary, given that the observer does not know the seed or any state values.

When observed from within the DRBG boundary, each of the specified DRBGs (see Annex C) provides enhanced backward secrecy.

In the case of a DRBG, enhanced forward secrecy depends on the insertion of sufficient entropy during a generation process to meet the security requirements for the security strength to be supported by the DRBG. This entropy may be inserted during a reseeding process or provided as additional input from an additional randomness source. If this entropy is provided during each generation request, then full forward secrecy is usually provided.

NOTE 2 For example, if a DRBG is expected to provide 128 bits of security, then to have the full forward secrecy, the seed provides 128 bits of entropy and, for each iteration of the DRBG, an additional input is made that also contains at least 128 bits of entropy.

Providing entropy during each generation request is not necessarily practical for many applications. However, user input with even a small amount of entropy provides some degree of forward secrecy. It can be appropriate for an application to require additional input with high entropy for critical applications (e.g. the generation of digital signature keys).

With respect to backward secrecy, a properly implemented DRBG is instantiated using a secret seed. The seed is used to determine the initial state of the DRBG. The output of the DRBG is some function of the internal state, and the internal state is updated during each request for bits. If the seed or any state becomes known, prior outputs can be determined unless a cryptographically strong one-way function is used in the DRBG design to transition from one internal state to the next internal state.

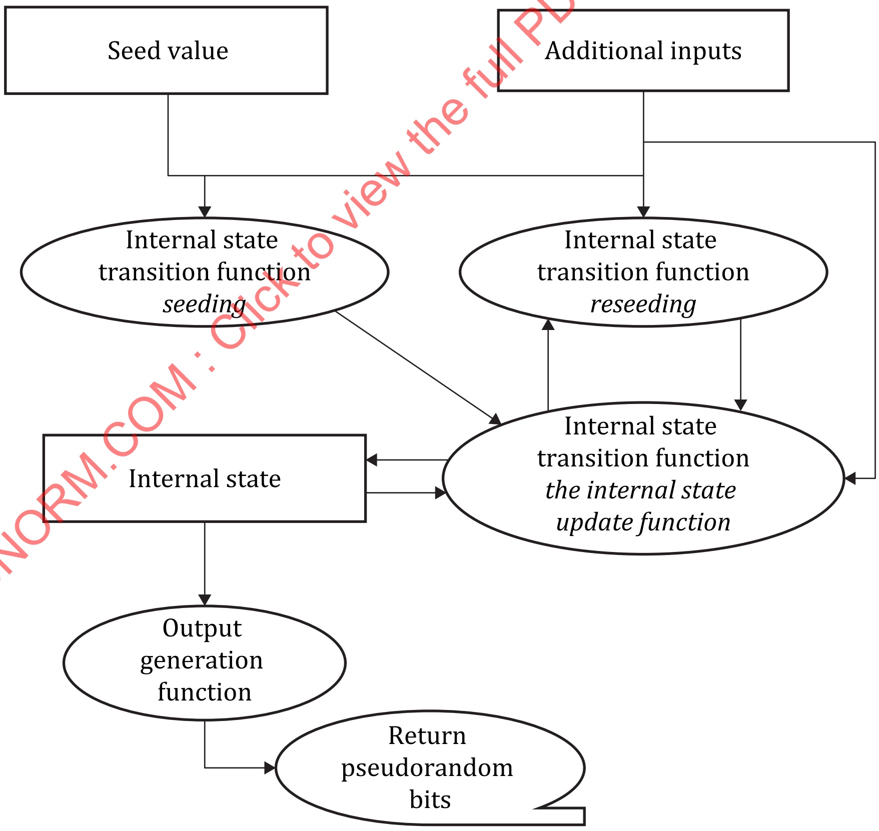
Seed value
Additional inputs
Internal state transition function *seeding*
Internal state transition function *reseeding*
Internal state
Internal state transition function *the internal state update function*
Output generation function
Return pseudorandom bits

Figure 5 — DRBG model

9.3 DRBG randomness source

9.3.1 Primary randomness source for a DRBG

The primary randomness source input for a DBRG is a seed value. The seed value shall contain sufficient randomness to support the DRBG's intended security strength and shall be input to the DRBG prior to requesting pseudorandom bits from the DRBG. For more details about the requirements of the source of the seed value, see 9.3.2.

The seed, seed value size and the entropy (i.e. randomness) of the seed shall be selected to minimize the probability that the sequence produced by one seed significantly matches the sequence produced by another seed, and to reduce the probability that the seed can be guessed or exhaustively tested. Since this document does not require full entropy for a seed but does require sufficient randomness, the length of the seed value used to construct the seed can be greater than the security strength specified in order to accommodate the needed randomness. That is, the length of the seed value shall, at minimum, be the number of bits in the specified security strength. However, the length should be more than the minimum in order to increase assurance of sufficient randomness and address any concerns of possible reuse of a seed. It is important to note that using the minimum length is acceptable only if the seed value is generated by a full entropy source. As the seed value is generally of variable size, the specification of the DRBG algorithms make this assumption, although a specific implementation can be optimized to support a specific length.

Randomness analysis for a seed is a critical component of the security assurance associated with the DRBG. The importance of this analysis is most readily demonstrated when something goes wrong. For example, if a security strength of 128 bits is desired and is believed to be achieved but only 60 bits are actually achieved, this amount of randomness is exhausted quite easily by an adversary. Even if the actual randomness is 80 bits, a determined adversary can still exhaust the randomness at a reasonable cost.

The generation or entry of randomness into a DRBG using an insecure method can result in voiding the intended security assurances. To ensure unpredictability, care should be exercised in obtaining and handling seeds. The seed and its use by a DRBG shall be generated and handled as follows.

1) Seed construction: A seed shall include a seed value and should include a personalization string (see 9.4 for further information on personalization strings). The combination of the seed value and the optional personalization string is also called the seed material. A derivation function shall be used to uniformly distribute the randomness across the seed or to adjust its length (without reducing the amount of randomness in the seed) whenever a personalization string is used. Whether or not the personalization string is present, the resulting seed shall be unique.

NOTE 1 Unique personalization strings can prevent an attacker from identifying the DRNG, inhibit side channel analysis, and be the last resort in a catastrophic failure scenario.

2) Seed use: DRBGs may be used to generate both secret and public information. In either case, the seed shall be kept secret. A single instantiation of a DRBG should not be used to generate both secret and public values. Cost and risk factors shall be taken into account when determining whether different instantiations for secret and public values can be accommodated.

NOTE 2 If the DRBG can be used to provide enhanced backward and forward secrecy and if the implementation is resistant against side channel attacks, it is possible to ignore this security feature to not use a single instantiation of a DRBG to generate both secret and public values without loss of security.

A seed that is used to initialize one instantiation of a DRBG shall not be intentionally used to reseed the same instantiation or used as a seed for another DRBG.

3) Seed randomness: The seed value for the seed shall contain sufficient randomness for the desired security strength, and the randomness shall be distributed across the seed. It is possible that a consuming application can be concerned about collision resistance. In order to accommodate possible collision concerns, a seed shall have randomness that is equal to or greater than 128 bits or the required security strength for the consuming application; whichever is greater [i.e. entropy ≥ max (128, *security_strength*)]. If a selected DRBG and the seed are not able to provide the strength required by the consuming application, then a different DRBG shall be used.

4) Seed value size: The minimum size of the seed value depends on the selected DRBG, the security strength required by the consuming application and the randomness source. The size of the seed value shall be at least equal to the required security strength of the DRBG in-bits and can be larger, depending on the randomness source [see item 2) in this subclause]. For example, if 160 bits of randomness are required, the quality of the randomness source can require a seed value size of 240 bits or more to achieve the 160 bits of randomness.

5) Seed privacy: Seeds and seed values shall be handled in a manner consistent with the security required for the target data. For example, if the only secrets in a cryptographic system are the keys, then the seeds used to generate keys shall be treated as if they are keys.

6) Seed usage period: A DRBG seed shall have a specified usage period, after which it shall no longer be used. Seeds shall have a specified finite seedlife. The seed shall be updated periodically, or the seed shall be rendered inoperable by replacement after its seedlife. If seeds become known (i.e. the seeds are compromised), unauthorized entities can determine the DRBG output. The usage period may be given by a time span or a maximum number of outputs that may be generated with that same seed.

   In some implementations (e.g. smartcards), an adequate reseeding process is not possible, since a randomness source is possibly no longer available for reseeding. In these cases, the best policy can be to replace the DRBG, thus obtaining a new seed in the process (e.g. obtain a new smart card). In other applications, reseeding is an appropriate design choice. Reseeding (i.e. replacement of one seed with a new seed) is a means of recovering the secrecy of the output of the DRBG if a seed becomes known to an adversary. Periodic reseeding is a good countermeasure to the potential threat that the seeds and DRBG output become compromised. However, the result from reseeding is only as good as the quality of the new seed value provided for reseeding. Generating too many outputs from a seed (and other input information) can provide sufficient information for an adversary to successfully predicting future outputs. Periodic reseeding reduces security risks, reducing the likelihood of a compromise of the target data that is protected by cryptographic mechanisms that use the DRBG.

   Reseeding of the DRBG shall be performed in accordance with the specification for the given DRBG. When a new seed is obtained during reseeding, the new seed should be checked (where possible) to ensure that two consecutive seeds are not identical. More than one seed shall not be saved by the DRBG. A seed shall not be saved in its original form but shall be transformed by a one-way process. When a new seed is generated and compared to the "old" seed (i.e. the transformed old seed), the transformed new seed shall replace the old, transformed seed in memory. The old, transformed seed shall be destroyed. If the new seed is determined to be identical to the old seed, another new seed shall be generated.

7) Seed separation: It is recommended that when resources permit (e.g. storage capacity), different DRBG instantiations be used for the generation of different types of random data. Each instantiation shall be initialized with a different randomly generated seed value and have its own internal state, whether using the same or different DRBG techniques. For example, the instantiation used to generate public values should be different than the instantiation used to generate secret values. The instantiation used to generate asymmetric key pairs should be different than the instantiation used to seed other DRBGs, which should, in turn, be different than the instantiation used by the same (or a different) DRBG technique to generate symmetric keys. The instantiation used by a DRBG technique to generate random challenges should be different than the instantiation used by the same (or a different) DRBG technique to generate PINS or passwords. However, the amount of seed separation (i.e. the number of instantiations and DRBG techniques) is a cost/benefit decision.

NOTE 3 Requirement 11 in 5.2 states that a DRBG is required to provide enhanced backward secrecy. If enhanced forward secrecy is invoked (i.e. by inserting sufficient new entropy into the DRBG) and if the implementation is resistant against side-channel attacks, the random bits generated after the insertion of the new entropy can be used for different types of random data than the random bits generated before the new entropy was inserted without loss of security.

9.3.2 Generating seed values for a DRBG

The seed value used in the seed for an approved (target) DRBG shall be produced in one of three ways.

1) The seed value for the target DRBG shall be produced by an approved NRBG that produces output at a sufficient entropy rate (i.e. with full entropy).

2) The seed value for the target DRBG shall be produced by an approved (source) DRBG that produces output at a security strength equal to or greater than the security strength intended for the target DRBG. The source DRBG shall also have been seeded in accordance with these seed-generation requirements. Hence, a chain of DRBGs can be envisioned in which each DRBG produces seed values for the next DRBG in the chain. However, this chain shall always begin with either an approved NRBG or an approved DRBG with an internal entropy source.

3) The seed value shall be generated by an appropriate entropy source. This entropy source may produce dependent bits. If it is produced in this way, then the developer shall assess the rate of entropy production of the source and ensure that the source meets all the requirements of entropy sources found in 6.2.2, 8.3 and 8.8.4.

9.3.3 Additional randomness sources for a DRBG

The operation of a DRBG may also include one or more additional randomness sources. An additional randomness source can be useful for a variety of reasons.

In addition to the required seed value, a DRBG may obtain randomness from a non-deterministic randomness source (e.g. an entropy source or NRBG) or another DRBG after initialization. In this case, the DRBG is called a hybrid DRBG (see 9.3.4 for further information).

Despite the additional sources of randomness providing additional unpredictability to the output of a DRBG, the security of the DRBG shall rest solely upon the primary randomness source.

The advantage of using an entropy source or NRBG as an additional randomness source is that it allows the output of the DRBG to be non-deterministic and can help prevent cryptanalysis and/or add security features such as enhanced forward and backward secrecy.

9.3.4 Hybrid DRBGs

A DRBG is hybrid if it accepts external input after initialization (e.g. by reseeding or accepting additional input); otherwise, it is a pure DRBG.

The extra functional requirements of a hybrid DRBG are as follows.

1) An adversary shall not be able to predict the next bit with a probability significantly greater than one half, even if the attacker has complete control over non-deterministic input (e.g. from an entropy source or NRBG).

2) No adversary shall be able to recover any information about the non-deterministic input by observing the output of the RBG.

3) No unauthorized person shall be able to manipulate or influence the non-deterministic input.

9.4 Additional inputs for a DRBG

The operation of a DRBG may include optional additional inputs. Additional input information can be acquired by a DRBG during the instantiation, generation, and reseeding processes. This information includes the input parameters when the DRBG is called by the consuming application and any additional input that can be public. Additional inputs shall not weaken the RBG.

Depending on the DRBG, time-variant information can also be required, e.g. a counter or a date/time value.

DRBGs allow for the use of an optional personalization string during instantiation. A personalization string is used in conjunction with a seed value to produce a seed. Examples of data that can be included in a personalization string include a product and device number, user identification, date and timestamp, IP address, or any other information that helps to differentiate DRBGs.

The functional requirements for additional inputs are as follows.

1) When a counter is used by a DRBG, it shall not repeat during the “instance” of the DRBG. When the DRBG is initialized with a new seed, the counter can be set to a fixed value (e.g. set to 1) but shall be updated for each state of the DRBG and shall not repeat.

2) When a date/time value is used by a DRBG, it shall not repeat. Whenever a date/time value is requested by a DRBG, either a different date/time value shall be used than was previously used, or another technique shall supplement the date/time value to provide uniqueness (e.g. a counter is concatenated to the date/time value).

3) When a personalization string is used, it shall be specified to be unique for all instantiations of the same DRBG type.

4) A DRBG shall remain secure even if the adversary has full control over the additional inputs to the DRBG. That is, even if the attacker can adaptively choose the values of the additional inputs, the attacker should be unable to predict the next bit produced by the RBG with a probability significantly greater than one half.

9.5 Internal state for a DRBG

The internal state is the memory of the DRBG. It consists of all of the parameters, variables and other stored values that the DRBG uses or acts upon. The internal state includes values that are acted upon by the internal state transition function between requests, keys used during each call, additional input that has been obtained while a request is serviced and time-variant parameters used by the DRBG. The internal state is dependent on the specific DRBG and includes all information that is required to produce the pseudorandom bits from one request to the next. Some portion of the internal state shall be changed by the internal state transition function.

The internal state can be thought of as a combination of a working state (which is potentially updated during every execution), a secret parameter (which is constant during every execution and only updated periodically), and administrative information (fixed information for the instantiation, such as the security strength of the DRBG).

The functional requirements for the internal state are as follows.

1) A DRBG shall be instantiated prior to the generation of output by the DRBG. During instantiation, an initial state for the DRBG is derived, in part, from a seed. The DRBG instantiation may be reseeded at any time if a reseed capability has been included. The state of a DRBG includes information that is acted upon, and optionally, keys used by the generator.

2) The internal state shall be completely contained within the DRBG boundary.

3) The internal state shall be protected at least as well as the intended use of the output by the consuming application.

4) If there are values that the secret parameter should not take (e.g. “weak” cryptographic keys), then the secret parameter shall be tested to make sure that these secret parameter values are not used.

5) The secret parameter, if it exists, shall be replaced periodically unless the DRBG is terminated.

9.6 Internal state transition function for a DRBG

The internal state transition function updates the internal state using one or more algorithms. The algorithms used and the method of altering the internal state depends on the specific DRBG.

The DRBGs in this document have three separate internal state transition functions:

1) prior to the initial use of the DRBG, a seed value and all initial input are obtained. The seed value and initial input are used to determine the initial state of the DRBG;

2) the internal state shall be updated either after the output bits are determined by the OGF or when insufficient entropy is not yet available in a pool of bits from which the OGF can extract output; and

3) if the seed shall be replaced, then go to step 1 to determine a new internal state. Otherwise, combine new seed material with the current internal state values to determine a new internal state.

The functional requirements of the internal state transition function are as follows.

a) The internal state transition function shall allow for known-answer testing when required.

b) A DRBG shall transition between states on demand (i.e. when the generator is requested to provide new pseudorandom bits). A DRBG may also be implemented to transition in response to external events (e.g. system interrupts) or to transition continuously (e.g. whenever time is available to run the generator). Additional unpredictability is introduced when the generator transitions between states continuously or in response to external events. However, when the DRBG transitions from one state to another between requests, it can be necessary to perform reseeding.

c) The internal state transition functions shall achieve backward secrecy through appropriate use of a one-way function, such as a cryptographic hash-function.

d) The internal state transition functions shall have the property that all of the bits in the working state and any new input from the randomness source influence the contents of the next internal state.

e) The operation of the internal state transition function shall be protected against observation and analysis via power consumption, timing, radiation emission, or other side channel attacks, as appropriate. The values that the internal state transition function operates on (internal state, secret parameter, and any newly provided input from a randomness source) are the critical values upon which the confidentiality of future random output is based. Side channel analysis can potentially defeat this confidentiality.

f) It shall be ensured that the internal state transition function that updates the internal state has the property that the range of its possible values does not degenerate after repeated calls to it without intermediate reseeding.

9.7 Output generation function for a DRBG

The output generation function of a DRBG produces pseudorandom bits that are a function of both the internal state of the DRBG and any input that is introduced while the internal state transition function is operating. These pseudorandom bits are deterministic with respect to the content of the internal state. Any formatting of the bits prior to output is determined by a particular implementation.

The functional requirements for the output generation function are as follows.

1) The output generation function shall allow for known-answer testing when required.

2) A DRBG shall not provide output until a seed with sufficient entropy has been incorporated into the internal state.

3) A DRBG requiring key(s) shall not provide output until the key(s) is available.

4) The output generation function shall not leak any information about the internal state that can potentially enable future output to be compromised. Preferably, the output generation function should not be able to be inverted to reveal any information about the internal state. That is, knowledge of the random output produced by the output generation function should not reveal any information about the input to the function (i.e. the output generation function should be a one-way function).

5) The output generation function shall produce output using the DRBG's internal state.

6) The output generation function shall not reuse bits from the internal state.

9.8 Health tests for a DRBG

9.8.1 DRBG health tests overview

Although not actually shown in Figure 5, the DRBG intrinsically contains mechanisms to measure its health.

A DRBG shall be designed to permit testing that ensures that the generator is correctly implemented and continues to operate correctly. A test function shall be available for this purpose. The test function shall also allow the insertion of predetermined values of the input information in order to test for expected results (known-answer tests). If any test fails, the DRBG shall enter an error state and output an error indicator. The DRBG shall not perform any random output generation while in an error state and all output shall be inhibited.

NOTE Error states can include "hard" errors that indicate an equipment malfunction requiring maintenance, service, repair, or replacement of the DRBG, or can include recoverable "soft" errors requiring an initialization or resetting of the DRBG. Recovery from error states is possible except for those caused by hard errors that require maintenance, service, repair, or replacement of the DRBG.

9.8.2 DRBG health test

A DRBG shall perform health tests to ensure that it continues to function properly. Health tests of the RBG functionality shall be performed when the DRBG is powered up, on demand (e.g. upon application request or when resetting, rebooting, or power recycling), and under various conditions, typically when a particular function or operation is performed (i.e. conditional tests). Some health tests shall be operated continuously. An RBG may optionally perform other health tests for DRBG functionality in addition to the tests specified in this document.

All data output from the DRBG shall be inhibited while these tests are performed. The results from known-answer tests shall not be output as random bits. However, the bits used during other types of tests may be used as outputs if those tests do not detect errors.

When a DRBG fails a health test, it shall enter an error state and output an error indicator. The DRBG shall not perform any DRBG operations while in the error state, and no data shall be output when an error state exists. When in an error state, user intervention (e.g. power cycling, restart of the DRBG) shall be required to exit the error state.

9.8.3 DRBG deterministic algorithm test

This test shall be performed for a DRBG algorithm. A known-answer test shall be conducted at power up, on demand, and can be conducted at periodic intervals. A known-answer test involves operating the algorithm on data for which the correct output is already known and determining if the calculated output equals the expected output (the known answer). The test fails if the calculated output does not equal the known answer. In this case, the DRBG shall enter an error state and output an error indicator.

9.8.4 DRBG software/firmware integrity test

This test applies to any DRBG containing software or firmware. A software/firmware integrity test using an authentication technique shall be applied to all software and firmware residing in the DRBG when the DRBG is powered up in order to determine code integrity. Authentication techniques may include message authentication codes or digital signatures using known algorithms, or error detection codes (EDCs) when performed on code that only resides within a cryptographic boundary. This test fails if the calculated result does not equal the previously generated result. In this case, the DRBG shall enter an error state and output an error indicator.

For more details, refer to ISO/IEC 19790.

9.8.5 DRBG critical functions test

All other functions that are critical to the secure operation of the DRBG shall be tested during power-up and on demand. Critical DRBG functions that are performed under certain specific conditions shall also be tested when those conditions arise.

9.8.6 DRBG software/firmware load test

This test shall be performed by DRBGs that contain software or firmware. A cryptographic mechanism using an approved authentication technique (e.g. using an approved authentication code, digital signature algorithm, or HMAC) shall be applied to all software and firmware [e.g. within electrically-erasable programmable read-only memory (EEPROM), RAM, and field programmable gate arrays] that can be externally loaded into a DRBG. This test shall verify the authentication code or digital signature. A calculated result shall be compared with a previously generated result. This test fails if the two results do not match. In this case, the DRBG shall enter an error state and output an error indicator.

For more details, refer to ISO/IEC 19790.

9.8.7 DRBG manual key entry test

When security information is manually entered into a DRBG (e.g. a seed value or key), the security information shall include an error detection code (EDC) or duplicate entries shall be used in order to verify the accuracy of the security information. An EDC shall be at least 16 bits in length. A DRBG shall verify the entered data using the EDC or verify that the duplicate entries match, as appropriate. If the calculated EDC does not equal the EDC that was entered or the duplicate entries do not match, then the test fails. In this case, the DRBG shall enter an error state and output an error indicator.

9.8.8 Continuous tests on noise sources in entropy sources

Continuous health tests shall be run on the outputs of a noise source within an entropy source while the noise source is operating and before any conditioning of the noise source output. Continuous tests focus on the noise source behavior and aim to detect failures as the noise source produces outputs. The purpose of continuous tests is to allow the entropy source to detect many kinds of failures in its underlying noise source. These tests are run continuously on all digitized samples obtained from the noise source, and so tests shall have a very low probability of raising a false alarm during the normal operation of the noise source. In many systems, a reasonable false positive probability makes it extremely unlikely that a properly functioning device indicates a malfunction, even in a very long service life. Continuous tests are resource-constrained, meaning their ability to detect noise source failures is limited. As a result, they are usually designed so that only gross failures are likely to be detected.

It is important to note that continuous health tests operate over a stream of values. These sample values may be output from the entropy source as they are generated and (optionally) processed by a conditioning component. It is not necessary to inhibit output from the noise source or entropy source while running the test. It is important to understand that this can result in poor entropy source outputs for a time, since the error is only signaled once significant evidence has been accumulated, and it is possible that these values have already been output by the entropy source. As a result, it is important that the false positive probability be set to an acceptable level.

9.9 Additional requirements for DRBG keys

In addition to the functional requirements levied upon the DRBG components, other requirements are also imposed on the implementation and use of a DRBG. These requirements are associated with any key that is used by a given DRBG.

Some DRBGs require the use of one or more keys. When not explicitly prohibited, these keys may be provided from an external source (i.e. from an appropriate source outside the DRBG boundary), or the DRBG may be designed to generate keys from seed material. The use of externally provided keys can be appropriate, for example, in low-risk applications with memory constraints (e.g. smart cards), when the generation of sufficient seed material for keys is impractical (e.g. the source of sufficient entropy is too costly), or the quality of the DRBG's entropy source is questionable but high-quality keys can be obtained outside the DRBG boundary.

A key and its use in a DRBG shall conform to the following:

1) Key use: Keys shall be used as described in a specific DRBG. A DRBG requiring a key(s) shall not provide output until the key(s) is available.

2) Key entropy: The entropy for the combination of keys shall equal at least the required security strength of the consuming application. For example, when 128 bits of security are required by an application, the key(s) shall have at least 128 bits of entropy.

3) Key size: Key sizes shall be selected to support the desired security strength of the consuming application.

4) Keys determined from a seed: A key determined from a seed shall be independent of the rest of the initial input that was determined by that seed. If multiple keys are used by a DRBG, as opposed to the same key used in multiple places, then each key shall be independent of all other keys.

   a) For DRBGs that determine a key from the same seed as an initial value and any other keys (i.e. a single seed is used to determine all initial inputs for the DRBG, including keys), the seed shall have entropy that is equal to or greater than the intended security strength of the DRBG instantiation.

   b) For DRBGs that use multiple seed values to determine a DRBG instance, each seed value shall be used to determine a different part of the initial input (e.g. the initial value for the DRBG and each distinct key shall be determined from different seed values). The combination of all seed values shall have entropy that is equal to or greater than the required strength of the consuming application.

5) Keys provided from an external source: Unless keys generated externally have an amount of entropy equal to or greater than the security strength to be provided by the DRBG, the security of the DRBG cannot be guaranteed.

6) Rekeying: Rekeying (i.e. replacement of one key with a new key) is a means of recovering the secrecy of the output of the DRBG if a key becomes known. Periodic rekeying is a good countermeasure to the potential threat that the keys and DRBG output become compromised. However, the result from rekeying is only as good as the randomness source (e.g. NRBG or chain of DRBGs) used to provide the new key. In some implementations (e.g. smartcards), it is possible that there is no adequate rekeying process and rekeying can actually reduce security. In these cases, the best policy can be to replace the DRBG, obtaining a new key in the process (e.g. obtain a new smart card).

   Generating too many outputs using a given key can provide sufficient information for successfully predicting future outputs. Periodic rekeying reduces security risks, reducing the likelihood of a compromise of the target data that is protected by cryptographic mechanisms that use the DRBG.

7) Key life: Keys shall have a specified finite key life. Keys shall be updated (i.e. replaced) periodically. Expired keys, or keys from which updated or new values were derived, shall be destroyed. If keys become known (e.g. the seeds are compromised), unauthorized entities can possibly determine the DRBG output.

8) Key separation: A key used by a DRBG shall not intentionally be used for any purpose other than random bit generation. Different instances of a DRBG shall use different keys.

Annex A
(normative)

Combining RBGs

This annex specifies different ways to combine RBGs, which shall be followed when RBGs are combined. A combination of RBGs shall follow a “no worse than” paradigm; that is, the proposed combination of RBGs is "no worse than" an approved RBG (and, in theory, is intended to be better). This is a conservative way to combine RBGs.

An RBG may be combined with another RBG if it can be shown that the RBGs are not correlated with each other. For example, this can be because they use either a different seed, or different algorithm, or both. If two different algorithms were combined, this would address possible concerns of a discovery in the future of a cryptanalytic attack on one of the algorithms.

An approved RBG may be combined with an unapproved RBG not specified in this document if it can be shown that the two RBGs are not correlated and if it can be shown that if the second RBG degrades to a constant (that is, zero entropy). Then, the output of the combined RBG is still the output of an approved RBG. As an example, if the two RBG output streams are combined using an exclusive OR (XOR) operation, then this condition is satisfied.

Combining an approved RBG with an unspecified RBG can be done to allow the advantages of the unspecified RBG to be gained, which were thought to be superior to approved alternatives in some quality that is not specified in this document. It also can be done when one forum requires the use of one method and another forum requires the use of another method.

EXAMPLE 1 An NRBG with a physical entropy source uses a properly initialized DRBG to mix a “pool” of bits obtained from the physical source.

EXAMPLE 2 A generator that is composed of a “complete” NRBG that provides input to a “complete” DRBG. It is not possible to influence the flow of bits from the NRBG to the DRBG except, perhaps, for health testing or product validation.

Annex B
(normative)

Conversion methods for random number generation

B.1 Techniques for generating random numbers

This document is concerned with the generation of sequences of random bits. In some cryptographic applications, sequences of random numbers are required $(a_0, a_1, a_2, \ldots)$ where:

1) each integer a_i satisfies $0 \le a_i \le r-1$, for some positive integer r (the range of the random numbers);

2) the equation $a_i = s$ holds with probability almost exactly $1/r$, for any $i \ge 0$ and for any s ($0 \le s \le r-1$); and

3) each value a_i is statistically independent of any set of values a_j ($j \ne i$).

This annex specifies six techniques by which such sequences of random numbers can be generated from sequences of random bits.

If the range of the number required is not $0 \le a_i \le r-1$ but $a \le a_i \le b$, then a random number in this range can be obtained by computing $a_i + a$, where a_i is a random number in the range $0 \le a_i \le b-a$.

B.2 The simple discard method

Let m be the unique positive integer satisfying $2^{m-1} < r \le 2^m$ (m is uniquely defined by the choice of r). The method to generate the random number a is as follows.

1) Use the RBG to generate a sequence of m random bits, $(b_0, b_1, \ldots, b_{m-1})$.

2) Let $c = \sum_{i=0}^{m-1} 2^i b_i$.

3) If $c < r$ then put $a = c$, else discard c and go to step 1.

NOTE The ratio $r/2^m$ is a measure of the efficiency of the technique, and this ratio always satisfies $0{,}5 < r/2^m \le 1$. If $r/2^m$ is close to 1, then the above method is simple and efficient. However, if $r/2^m$ is close to 0,5, then the above method is less efficient, and the more complex method in B.3 is preferable.

B.3 The complex discard method

Choose a small positive integer t, and then let m be the unique positive integer satisfying $2^{m-1} < r^t \le 2^m$ (m is uniquely defined by the choices of r and t). This method generates a sequence of t random numbers $(a_0, a_1, \ldots, a_{t-1})$ in the following way.

1) Use the RBG to generate a sequence of m random bits, $(b_0, b_1, \ldots, b_{m-1})$.

2) Let $c = \sum_{i=0}^{m-1} 2^i b_i$.

3) If $c < r^t$ then let $(a_0, a_1, \ldots, a_{t-1})$ be the unique sequence of values satisfying $0 \le a_i \le r-1$ such that $c = \sum_{i=0}^{t-1} r^i a_i$, else discard c and go to step 1.

NOTE 1 The ratio $r^t/2^m$ is a measure of the efficiency of the technique, and this ratio always satisfies $0{,}5 < r^t/2^m \leq 1$. Hence, given r, it is preferable to choose t so that $r^t/2^m$ is sufficiently close to 1. For example, if $r=3$, then choosing $t=3$ means that $m=5$ and $r^t/2^m = 27/32 \approx 0{,}84$, and choosing $t=5$ means that $m=8$ and $r^t/2^m = 243/256 \approx 0{,}95$.

NOTE 2 The complex discard method coincides with the simple discard method when $t=1$.

B.4 The simple modular method

Let m be the unique positive integer satisfying $2^{m-1} < r \leq 2^m$, and let l be a security parameter. The method to generate the random number a is as follows.

1) Use the RBG to generate a sequence of $m+l$ random bits, $(b_0, b_1, \ldots, b_{m+l-1})$.

2) Let $c = \sum_{i=0}^{m+l-1} 2^i b_i$.

3) Let $a = c \bmod r$.

NOTE Unlike the previous two methods, the simple modular method is guaranteed to terminate after one execution. Unfortunately, the probability that $a=s$ for any particular value of s $(0 \leq s \leq r-1)$ is not exactly $1/r$. However, for a large enough value of l the difference between the probability that $a=s$ for any particular value of s and $1/r$ are negligible.

B.5 The complex modular method

Choose a small positive integer t, and then let m be the unique positive integer satisfying $2^{m-1} < r^t \leq 2^m$ (m is uniquely defined by the choices of r and t). Let l be a security parameter. This method generates a sequence of t random numbers $(a_0, a_1, \ldots, a_{t-1})$ in the following way.

1) Use the RBG to generate a sequence of $m+l$ random bits, $(b_0, b_1, \ldots, b_{m+l-1})$.

2) Let $c = \sum_{i=0}^{m+l-1} 2^i b_i \bmod r^t$.

3) Let $(a_0, a_1, \ldots, a_{t-1})$ be the unique sequence of values satisfying $0 \leq a_i \leq r-1$ such that $c = \sum_{i=0}^{t-1} r^i a_i$.

NOTE 1 As with the simple modular method, the complex modular method does not give a truly uniform distribution of numbers, but the difference between the actual distribution of numbers and the uniform distribution is negligible for a large enough security parameter.

NOTE 2 The complex modular method coincides with the simple modular method when $t=1$.

B.6 The simple partial discard method

Let m be the unique positive integer satisfying $2^{m-1} < r \leq 2^m$ (m is uniquely defined by the choice of r). The method to generate random number a is as follows.

1) Use the RBG to generate a sequence of m random bits, $(b_0, b_1, \ldots, b_{m-1})$.

2) Let $c = \sum_{i=0}^{m-1} 2^i b_i$.

3) If $c < r$ then put $a = c$, else

a) let d be the unique integer such that $2^{m-d-1} \leq c \oplus (r-1) < 2^{m-d}$,

b) use the RBG to generate a sequence of $d+1$ random bits, $(b_m, b_{m+1}, \ldots, b_{m+d})$,

c) let $b_i = b_{m+i}$ for $0 \le i < d+1$ and $b_{d+1+i} = b_i$ for $0 \le i < m-d-1$ and go to step 2.

NOTE 1 In contrast to the simple discard method that discards all m random bits when $c \ge r$, the simple partial discard method discards $d+1$ ($\le m$) bits when $c \ge r$.

NOTE 2 The simple partial discard method gives a uniform distribution of numbers, i.e. the probability that $a = s$ for any particular value of s ($0 \le s \le r-1$) is exactly $1/r$.

NOTE 3 For any positive integer r, the expected number of random bits needed to generate a random number of the simple partial discard method is less than or equal to that of the simple discard method.

B.7 The complex partial discard method

Choose a small positive integer t, and then let m be the unique positive integer satisfying $2^{m-1} < r^t \le 2^m$ (m is uniquely defined by the choices of r and t). This method generates a sequence of t random numbers $(a_0, a_1, \ldots, a_{t-1})$ in the following way.

1) Use the RBG to generate a sequence of m random bits, $(b_0, b_1, \ldots, b_{m-1})$.

2) Let $c = \sum_{i=0}^{m-1} 2^i b_i$.

3) If $c < r$ then let $(a_0, a_1, \ldots, a_{t-1})$ be the unique sequence of values satisfying $0 \le a_i \le r-1$ such that $c = \sum_{i=0}^{t-1} r^i a_i$, else

   a) let d be the unique integer such that $2^{m-d-1} \le c \oplus \left(r^t - 1\right) < 2^{m-d}$,

   b) use the RBG to generate a sequence of $d+1$ random bits, $(b_m, b_{m+1}, \ldots, b_{m+d})$,

   c) let $b_i = b_{m+i}$ for $0 \le i < d+1$ and $b_{d+1+i} = b_i$ for $0 \le i < m-d-1$ and go to step 2.

NOTE 1 Like the simple partial discard method, the complex partial discard method gives a uniform distribution of numbers.

NOTE 2 The complex partial discard method coincides with the simple partial discard method when $t = 1$.

Annex C
(informative)

Deterministic random bit generators

C.1 DRBG mechanism examples

This annex contains examples of deterministic random bit generator (DRBG) mechanisms that meet the requirements of this document. This is not a complete list. Other mechanisms are possible if they meet the requirements stated in the main body of this document.

The DRBGs within this annex are given in pseudocode.

NOTE Some pseudocode examples contained herein also appear in Reference [11].

C.2 DRBGs based on hash-functions

C.2.1 Introduction to DRBGs based on hash-functions

A hash DRBG is based on a hash-function that is non-invertible. DRBGs based on hash-functions are provided below.

The Hash_DRBG (...) specified uses any cryptographic hash-function as specified by ISO/IEC 10118-3 and ISO/IEC 29192-5 and may be used by applications requiring various security strengths, provided that the appropriate hash-function is used and sufficient entropy is obtained for the seed.

C.2.2 Hash_DRBG

C.2.2.1 Discussion

The design assumptions of the hash-function DRBG (Hash_DRBG) are as follows.

1) The outputs of the hash-function appear random if the inputs are different.

2) The seed contains an appropriate amount of entropy based on the bits of security desired, up to a maximum of the bit length of the output of the hash-function.

Hash_DRBG (...) employs a cryptographic hash-function specified by ISO/IEC 10118-3 and ISO/IEC 29192-5 that produces a block of pseudorandom bits using a seed (*seed*).

Optional additional input (*additional_input*) may be provided during each request of Hash_DRBG (...).

Hash_DRBG (...) has been designed to meet different security strengths (see Annex E) depending on the hash-function used.

The Hash_DRBG (...) generator requires the use of a hash-function at several points in the process, including the instantiation and reseeding processes. The same hash-function is used throughout. The hash-function should meet or exceed the desired security strength of the DRBG.

Table C.1 provides examples of cryptographic hash-functions specified by ISO/IEC 10118-3 and ISO/IEC 29192-5, illustrating security strength, required minimum entropy and seed length.

Table C.1 — Security strength table for hash-functions

Hash-function	**SHA-224**	**SHA-256**	**SHA-384**	**SHA-512**	**SHA3-224**	**SHA3-256**	**SHA3-384**	**SHA3-512**	**SHAKE128**	**SHAKE256**
Supported security strength	128, 192	128, 192, 256	128, 192, 256	128, 192, 256	128, 192, 224	128, 192, 256	128, 192, 256	128, 192, 256	128, 192, 256	128, 192, 256
Required minimum entropy	**max**(128, security strength)									
Seed length (*seedlen*)	440	440	888	888	440	440	888	888	440	888

C.2.2.2 Description

C.2.2.2.1 General

The instantiation and reseeding of Hash_DRBG (...) consists of obtaining a seed value with at least the amount of entropy requested by the consuming application. The seed value is used to derive a *seed*. The *seed* is used to derive elements of the initial *state*, which consists of:

1) a value (V) that is updated during each call to the DRBG;

2) a constant (C) that depends on the *seed*;

3) a counter (*reseed_counter*) that indicates the number of requests for pseudorandom bits since new *seed_value* was obtained during instantiation or reseeding;

4) the security *strength* of the DRBG instantiation;

5) the length of the *seed* (*seedlen*);

6) a *prediction_resistance_flag* that indicates whether or not enhanced forward secrecy capability is required by the DRBG; and

7) (optional) a transformation of the seed value using a one-way function for later comparison with a new seed value when the DRBG is reseeded; this value is present if the DRBG will potentially be reseeded; it may be omitted if the DRBG will not be reseeded.

The variables used in the description of Hash_DRBG (...) are:

additional_input — Optional additional input. It is $\leq 2^{35}$ bits.

C — A *seedlen*-bit constant that is calculated during the instantiation and reseeding processes.

data — The data to be hashed.

Get_entropy (*min_entropy*, *min_length*, *max_length*) — A function that acquires a string of bits from a randomness source. *min_entropy* indicates the minimum amount of entropy to be provided in the returned bits; *min_length* indicates the minimum number of bits to be returned; *max_length* indicates the maximum number of bits to be returned.

Hash (*a*) — A hashing operation on data *a* using a cryptographic hash-function specified by ISO/IEC 10118-3 and ISO/IEC 29192-5.

Hash_df (*seed_material*, *seedlen*) — A derivation function that hashes an input string and returns the number of bits according to the characteristics of the hash-function.

i — A temporary value used as a loop counter.

m — The number of iterations of the hash-function needed to obtain the requested number of pseudorandom bits.

max (*A*, *B*) — A function that returns either *A* or *B*, whichever is greater.

max_length — The maximum length of a string returned from the Get_entropy (...) function.

max_request_length — The maximum number of pseudorandom bits that may be requested during a single request. This value is implementation dependent. It is $\leq 2^{19}$ bits.

min_entropy — The minimum amount of *entropy* to be obtained from the randomness source and provided in the *seed*.

min_length — The minimum length of the *seed_value*.

Null — The null (i.e. empty) string.

outlen — The length of the hash-function output.

personalization_string — A *personalization string*. It is $\leq 2^{35}$ bits.

prediction_resistance_flag — A flag indicating whether or not enhanced forward secrecy requests should be handled.

prediction_resistance_flag = 1 = allow =
Allow_prediction_resistance,
prediction_resistance_flag = 0 = do not allow =
No_prediction_resistance.

prediction_resistance_request_flag — Indicates whether or not enhanced forward secrecy is required during a request for pseudorandom bits.

prediction_resistance_request_flag = 1 =
Provide_prediction_resistance,
prediction_resistance_request_flag = 0=
No_prediction_resistance.

pseudorandom_bits — The number of pseudorandom bits to be returned from Hash_DRBG (...) function.

requested_no_of_bits — The number of pseudorandom bits to be generated. It is $\leq 2^{19}$ bits.

requested_strength — The security strength to be associated with the requested pseudorandom bits.

reseed_counter — A count of the number of requests for pseudorandom bits since the instantiation or reseeding.

reseed_interval — The maximum number of requests for the generation of pseudorandom bits before reseeding is required. It is $\leq 2^{48}$ bits.

seed — The string of bits containing entropy that is used to determine the initial state of the DRBG during instantiation or reseeding.

seedlen — The length of the seed containing the required entropy.

seed_material The data that is used to create the seed.

seed_value The bits containing entropy that are used to determine the *seed_material* and generate a *seed*. It is $\leq 2^{35}$ bits.

state(*state_handle*) An array of states for different DRBG instantiations. A *state* is carried between DRBG calls. For the Hash_DRBG (...), the *state* for an instantiation is defined as *state*(*state_handle*) = {*V*, *C*, *reseed_counter*, *strength*, *prediction_resistance_flag*, *seedlen*}. A particular element of the *state* is specified as *state*(*state_handle*).*element*, e.g. *state*(*state_handle*).*V*.

state_handle A handle to the state space for the given instantiation.

status The status returned from a function call, where *status* = "Success" or a failure message.

strength The security strength for the DRBG.

V A value that is initially derived from the *seed*, but assumes new values based on optional additional input (*additional_input*), the pseudorandom bits produced by the generator (*pseudorandom_bits*), the constant (*C*) and the iteration count (*reseed_counter*).

w, *W* Intermediate values.

C.2.2.2.2 Instantiation of Hash_DRBG (...)

The following process or its equivalent is used to instantiate the Hash_DRBG (...) process.

Let Hash (...) be the cryptographic hash-function as specified in ISO/IEC 10118-3 and ISO/IEC 29192-5 and let *outlen* be the output length of that hash-function.

Instantiate_Hash_DRBG (...):

```
    Input: integer (requested_strength, prediction_resistance_flag), string personalization_string.

    Output: string status, integer state_handle.

    Process:

        1) If (requested_strength > the maximum security strength that can be provided for the hash-function), then Return (Failure message).

        2) Set the strength to one of the four security strengths.

            If (requested_strength ≤ 112), then strength = 112

                Else if (requested_strength ≤ 128), then strength = 128

                    Else if (requested_strength ≤ 192), then strength = 192

                        Else strength = 256.

        3) min_entropy = max(128, strength).

        4) min_length = max(outlen, strength).

        5) (status, seed_value) = Get_entropy (min_entropy, min_length, max_length).

        6) If (status ≠ "Success"), then Return (Failure message).
```

7) *seed_material* = *seed_value* || *nonce* || *personalization_string*.

8) *seed* = Hash_df (*seed_material*, *seedlen*).

NOTE 1 This step ensures that the entropy is distributed throughout the *seed*.

9) *V* = *seed*.

10) *C* = Hash_df ((`0x00` || *V*), *seedlen*).

11) Precede *V* with a byte of zeroes.

12) *reseed_counter* = 1.

13) *state*(*state_handle*) = {*V*, *C*, *reseed_counter*, *strength*, *prediction_resistance_flag*, *seedlen*}.

14) Return (“Success”, *state_handle*).

Get_entropy (...):

The specific details of the Get_entropy (...) process are left to the implementer. A high-level example is as follows:

Input: integer (*min_entropy*, *min_length*, *max_length*).

Output: string (*status*, *seed_value*).

Process:

1) Obtain a *seed_value* with *entropy* ≥ *min_entropy* and with the appropriate *length* from the appropriate randomness source, where *min_length* ≤ *length* ≤ *max_length*.

2) Return (“Success”, *seed_value*).

Hash_df (...):

Derivation functions are used during DRBG instantiation and reseeding to either derive state values or to distribute entropy throughout a bit-string. The hash-based derivation function hashes an input string and returns the requested number of bits. Let Hash (...) be the hash-function used by the DRBG and let *outlen* be its output length.

Input: string *input_string*, integer *no_of_bits*.

Output: bit-string *requested_bits*.

Process:

1) *temp* = the *Null* string.

2) $len = \left\lceil \frac{no\_of\_bits}{outlen} \right\rceil$.

3) *counter* = an 8-bit binary value represented in hexadecimal as `0x01`.

4) For *i* = 1 to *len* do

4.1 *temp* = *temp* || Hash (*counter* || *no_of_bits* || *input_string*).

NOTE 2 *no_of_bits* is represented as a 32-bit integer.

4.2 *counter* = *counter* + 1.

```
    5) requested_bits = Leftmost no_of_bits of temp.

    6) Return (requested_bits).
```

NOTE 3 If an implementation does not handle all four security strengths, then step 2 of Instantiate_Hash_DRBG(...) is modified accordingly.

NOTE 4 If no *personalization_string* is ever provided, then the *personalization_string* parameter in the input can be omitted, and step 7 of Instantiate_Hash_DRBG(...) becomes *seed_material* = *seed_value*.

NOTE 5 If an implementation does not require the *prediction_resistance_flag* as a calling parameter [i.e. the Hash_DRBG (...) routine in C.2.2.2.4 either always or never acquires new entropy in step 5], then the *prediction_resistance_flag* in the calling parameters and in the *state* [see step 12 of Instantiate_Hash_DRBG(...)] can be omitted.

C.2.2.2.3 Reseeding Hash_DRBG (...) instantiation

The following process or its equivalent is used to reseed the Hash_DRBG (...) process. Let Hash (...) be the cryptographic hash-function specified by ISO/IEC 10118-3 and ISO/IEC 29192-5 and let *outlen* be the output length of that hash-function.

Reseed_Hash_DRBG_Instantiation (...):

```
Input: integer state_handle, string additional_input.

Output: string status, where status = "Success" or a failure message.

Process:

    1) If the state indicated by state_handle is not available, then Return (Failure message).

    2) Get the appropriate state values, e.g. V = state(state_handle).V, strength = state(state_handle).
       strength, seedlen = state(state_handle).seedlen.

    3) min_entropy = max (128, strength).

    4) min_length = min_entropy.

    5) (status, seed_value) = Get_entropy (min_entropy, min_length, max_length).

    6) If (status ≠ "Success"), then Return (Failure message).

    7) seed_material = 0x01 || V || seed_value || additional_input.

    8) seed = Hash_df (seed_material, seedlen).

    NOTE This step combines the new seed_value with a fixed byte, the entropy present in V and with any
    additional_input provided; then distributed throughout the seed.

    9) V = seed.

    10) C = Hash_df ((0x00 || V), seedlen).

    11) Update the appropriate state values.

        11.1) state(state_handle).V = V.

        11.2) state(state_handle).C = C.

        11.3) state(state_handle).reseed_counter = 1.

    12) Return ("Success").
```

C.2.2.2.4 Generating pseudorandom bits using Hash_DRBG (...)

The following process or its equivalent is used to generate pseudorandom bits. Let Hash (...) be the cryptographic hash-function specified by ISO/IEC 10118-3 and ISO/IEC 29192-5 and let *outlen* be the output length of that hash-function.

Hash_DRBG (...):

```
Input:   integer (state_handle, requested_no_of_bits, requested_strength, prediction_resistance_request_flag), string additional_input.

Output: string status, bit-string pseudorandom_bits, where status = “Success” or a failure message.

Process:

    1)  If the state indicated by state_handle is not available, then Return (Failure message, Null).
    2)  Get the appropriate state values, e.g. V = state(state_handle).V, C = state(state_handle).C, reseed_counter = state(state_handle).reseed_counter, strength = state(state_handle).strength, seedlen = state(state_handle).seedlen, prediction_resistance_flag = state(state_handle).prediction_resistance_flag.
    3)  If (requested_no_of_bits > max_request_length), then Return (Failure message, Null).
    4)  If (requested_strength > strength), then Return (Failure message, Null).
    5)  If ((prediction_resistance_request_flag = Provide_prediction_resistance) and (prediction_resistance_flag = No_prediction_resistance)), then Return (Failure message, Null).
    6)  If ((reseed_counter > reseed_interval) OR (prediction_resistance_request_flag = Provide_prediction_resistance)) then:
        6.1  If reseeding is not available, then Return (Failure message, Null).
        6.2  status = Reseed_Hash_DRBG_Instantiation (state_handle, additional_input).
        6.3  If (status ≠ “Success”), then Return (status, Null).
        6.4  V = state(state_handle).V, C = state(state_handle).C, reseed_counter = state(state_handle).reseed_counter.
        6.5  additional_input = Null.
    7)  If (additional_input ≠ Null), then do
        7.1  w = Hash (0x02 || V || additional_input).
        7.2  V = (V + w) mod $2^{seedlen}$.
    8)  pseudorandom_bits = Hashgen (requested_no_of_bits, V).
    9)  H = Hash (0x03 || V).
    10) V = (V + H + C + reseed_counter) mod $2^{seedlen}$.
    11) reseed_counter = reseed_counter + 1.
    12) Update the changed values in the state.
        12.1    state(state_handle).V = V.
        12.2    state(state_handle).reseed_counter = reseed_counter.
    13) Return ("Success", pseudorandom_bits).
```

Hashgen (...):

Input: integer *requested_no_of_bits*, bit-string *V*.

Output: bit-string *pseudorandom_bits*.

Process:

```
1) $m = \left\lceil \frac{requested\_no\_of\_bits}{outlen} \right\rceil$.
2) $data = V$.
3) $W$ = the Null string.
4) For $i = 1$ to $m$
    4.1 $w_i$ = Hash ($data$).
    4.2 $W = W \,||\, w_i$.
    4.3 $data = (data + 1) \bmod 2^{seedlen}$.
5) pseudorandom_bits = Leftmost requested_no_of_bits of $W$.
6) Return (pseudorandom_bits).
```

NOTE 1 If an implementation never requests *additional_input*, then the *additional_input* input parameter and step 6 of Hash_DRBG can be omitted.

NOTE 2 If an implementation does not require the *prediction_resistance_flag*, then the reference to the *prediction_resistance_flag* in Hash_DRBG can be omitted.

C.2.2.2.5 Inserting additional entropy into the state of Hash_DRBG (...)

Additional entropy may be inserted into the state of the Hash_DRBG (...) in four ways. Additional entropy may be inserted by:

1) calling the Reseed_Hash_DRBG_Instantiation (...) function at any time. This function always calls the implementation dependent function Get_entropy (...) for *min_entropy* = max (128, *strength*) new bits of entropy, which are added to the state.

2) utilizing the automatic reseeding feature of the DRBG. If the maximum number of updates for the state is reached, the DRBG calls the Reseed_Hash_DRBG_Instantiation (...) process.

3) setting the *prediction_resistance_flag* to Allow_prediction_resistance at instantiation. If set, any call to the DRBG to generate pseudorandom bits may include a request for enhanced forward secrecy, which in turn invokes a call to Get_entropy (...) before new pseudorandom bits are produced.

4) supplying additional input during any call to the DRBG for pseudorandom bits.

NOTE 1 For the Hash_DRBG, additional input provided during a generation request only affects *V*. An actual reseed affects both *V* and *C*.

NOTE 2 Frequent calls to the Get_entropy (...) function can cause severe performance degradation.

C.2.3 HMAC_DRBG

C.2.3.1 Discussion

HMAC_DRBG uses multiple occurrences of an approved keyed hash-function (see ISO/IEC 10118-3) as specified in ISO/IEC 9797-2. This DRBG mechanism uses the update function specified in C.2.3.2.2 and the HMAC function within the Update function as the derivation function during instantiation and reseeding. The same cryptographic hash-function is used throughout an HMAC_DRBG instantiation. The hash-function used meets or exceeds the security requirements of the DRBG.

C.2.3.2 Description

C.2.3.2.1 General

The instantiation and reseeding of HMAC_DRBG (...) consists of obtaining a *seed value* with the appropriate amount of entropy. The seed value is used to derive a *seed*, which is then used to derive elements of the initial *state* of the DRBG. The *state* consists of:

1) the value *V*, which is updated each time another *outlen-bits* of output are produced (where *outlen* is the number of output bits from the output block of the cryptographic hash-function);

2) the *outlen*-bit *Key*, which is updated at least once each time that the DRBG mechanism generates pseudorandom bits;

3) a counter (*reseed_counter*) that indicates the number of requests for pseudorandom bits since instantiation or reseeding;

4) the security *strength* of the DRBG instantiation; and

5) a *prediction_resistance_flag* that indicates whether or not an enhanced forward secrecy capability is required for the DRBG.

The variables used in the description of HMAC_DRBG (...) are:

additional_input
Optional additional input. It is ≤2^{35} bits.

data
The data to be hashed.

Get_entropy (*min_entropy*, *min_length*, *max_length*)
A function that acquires a string of bits from a randomness source. *min_entropy* indicates the minimum amount of entropy to be provided in the returned bits; *min_length* indicates the minimum number of bits to be returned; *max_length* indicates the maximum number of bits to be returned.

HMAC (*K*, *V*)
The keyed hash-function specified by ISO/IEC 9797-2 using the cryptographic hash-function selected for the DRBG mechanism, where *K* is the key to be used, and *V* is the input block.

Key
The key used to generate pseudorandom bits.

len (*x*)
A function that returns the number of bits in input string *x*.

max (*A*, *B*)
A function that returns either *A* or *B*, whichever is greater.

max_length
The maximum length of a string returned from the Get_entropy (...) function. It is ≤2^{35} bits.

max_request_length The maximum number of pseudorandom bits that may be requested during a single request. This value is implementation dependent. It is $\leq 2^{19}$ bits.

min_entropy The minimum amount of *entropy* to be obtained from the entropy source and provided in the *seed value*.

min_length The minimum length of the bit-string to be acquired containing entropy.

Null The null (i.e. empty) string.

outlen The length of the cryptographic hash-function output.

provided_data Input bit-string.

personalization_string A personalization string.

prediction_resistance_flag A flag indicating whether or not enhanced forward secrecy requests should be handled.

prediction_resistance_flag = 1 = allow =

Allow_prediction_resistance

prediction_resistance_flag = 0 = do not allow =

No_prediction_resistance.

prediction_resistance_request_flag Indicates whether or not enhanced forward secrecy is required during a request for pseudorandom bits.

prediction_resistance_request_flag = 1 =

Provide_prediction_resistance,

prediction_resistance_request_flag = 0=

No_prediction_resistance.

pseudorandom_bits The pseudorandom bits produced by the generator.

requested_no_of_bits The number of pseudorandom bits to be returned from HMAC_DRBG (...) function. It is $\leq 2^{19}$ bits.

requested_strength The security strength to be associated with the requested pseudorandom bits.

reseed_counter A count of the number of requests for pseudorandom bits since the instantiation or reseeding.

reseed_interval The maximum number of requests for the generation of pseudorandom bits before reseeding is required. It is $\leq 2^{48}$ bits.

seed_material The data that is used to create the seed.

seed_value The bits containing entropy that are used to determine the *seed_material* and generate a *seed*. It is $\leq 2^{35}$ bits.

state(*state_handle*) An array of states for different DRBG instantiations. A *state* is carried between DRBG calls. For the HMAC_DRBG (...), the *state* for an instantiation is defined as *state* (*state_handle*) = {*V*, *Key*, *reseed_counter*, *strength*, *prediction_resistance_flag* }. A particular element of the *state* is specified as *state*(*state_handle*).*element*, e.g., *state*(*state_handle*).*V*.

state_handle A handle to the state space for the given instantiation.

status The status returned from a function call, where *status* = “Success” or a failure message.

strength The security strength for the DRBG.

temp An intermediate value.

V A value in the *state* that is updated whenever pseudorandom bits are generated.

C.2.3.2.2 Internal function: The Update function

The HMAC_DRBG_Update function updates the internal state of HMAC_DRBG using the *provided_data*. It should be noted that for this DRBG mechanism, the HMAC_DRBG_Update function also serves as a derivation function for the instantiation and reseed functions.

Let HMAC be the keyed hash-function found in ISO/IEC 9797-2 using the cryptographic hash-function selected for the DRBG mechanism from ISO/IEC 10118-3 or ISO/IEC 29192-5.

HMAC_DRBG_Update (...):

Input: bit-string (*provided_data*, K, V).

Output: bit-string (K, V).

Process:

1) K = HMAC (K, V || `0x00` || *provided_data*).
2) V = HMAC (K, V).
3) If (*provided_data* = *Null*), then Return (K, V).
4) K = HMAC (K, V || `0x01` || *provided_data*).
5) V = HMAC (K, V).
6) Return (K, V).

C.2.3.2.3 Instantiation of HMAC_DRBG

The following process or its equivalent is used to instantiate the HMAC_DRBG (...) process.

Let HMAC (...) be the ISO/IEC keyed hash-function to be used and let *outlen* be the output length of the keyed hash-function used.

Instantiate_HMAC_DRBG (...):

Input: integer (*requested_strength*, *prediction_resistance_flag*), string *personalization_string*.

Output: string *status*, integer *state_handle*.

Process:

1) If (*requested_strength* > the maximum security *strength* that can be provided for the MAC algorithm), then Return (Failure message).

2) Set the strength to one of the four security strengths.

   If (*requested_strength* ≤ 112), then *strength* = 112

      Else if (*requested_strength* ≤ 128), then *strength* = 128

         Else if (*requested_strength* ≤ 192), then *strength* = 192

            Else *strength* = 256.

3) *min_entropy* = max(128, *strength*).

4) *min_length* = max(*outlen*, *strength*).

5) (*status*, *seed_value*) = Get_entropy (*min_entropy*, *min_length*, *max_length*).

6) If (*status* ≠ "Success"), then Return (Failure message).

7) *seed_material* = *seed_value* || *nonce* || *personalization_string*.

8) *Key* = `0x00 00…00`. Its length is *outlen*.

9) *V* = `0x01 01…01`. Its length is *outlen*.

10) (*Key*, *V*) = HMAC_DRBG_Update(*seed_material*, *Key*, *V*).

11) *reseed_counter* = 1.

12) *state*(*state_handle*) = { *V*, *Key*, *reseed_counter*, *strength*, *prediction_resistance_flag* }.

13) Return ("Success", *state_handle*).

C.2.3.2.4 Reseeding HMAC_DRBG (...) Instantiation

The following process or its equivalent is used to reseed the HMAC_DRBG (...) process, after it has been instantiated.

Reseed_HMAC_DRBG_Instantiation (...):

Input: integer *state_handle*, string *additional_input*.

Output: string *status*, where *status* = "Success" or a failure message.

Process:

1) If the state indicated by *state_handle* is not available, then Return (Failure message).

2) Get the appropriate *state* values, e.g. *V* = *state*(*state_handle*).*V*, *Key* = *state*(*state_handle*).*Key*, *strength* = *state*(*state_handle*).*strength*.

3) *min_entropy* = max (128, *strength*).

4) *min_length* = *min_entropy*.

5) (*status*, *seed_value*) = Get_entropy (*min_entropy*, *min_length*, *max_length*).

6) If (*status* ≠ “Success”), then Return (Failure message).

7) *seed_material* = *seed_value* || *additional_input*.

8) (*Key*, *V*) = HMAC_DRBG_Update(*seed_material*, *Key*, *V*).

9) Update the appropriate *state* values.

   a) *state*(*state_handle*).*V* = *V*.

   b) *state*(*state_handle*).*Key* = *Key*.

   c) *state*(*state_handle*).*reseed_counter* = 1.

10) Return (“Success”).

C.2.3.2.5 Generating pseudorandom bits using HMAC_DRBG (...)

The following process or its equivalent is used to generate pseudorandom bits.

HMAC_DRBG (...):

Input: integer (*state_handle*, *requested_no_of_bits*, *requested_strength*, *prediction_resistance_request_flag*), string *additional_input*.

Output: string *status*, bit-string *pseudorandom_bits*.

Process:

1) If the state indicated by *state_handle* is not available, then Return (Failure message, *Null*).

2) Get the appropriate *state* values, e.g. *V* = *state*(*state_handle*).*V*, *Key* = *state*(*state_handle*).*Key*, *reseed_counter* = *state*(*state_handle*).*reseed_counter*, *strength* = *state*(*state_handle*).*strength*, *prediction_resistance_flag* = *state*(*state_handle*).*prediction_resistance_flag*.

3) If (*requested_no_of_bits* > *max_request_length*), then Return (Failure message, *Null*).

4) If (*requested_strength* > *strength*), then Return (Failure message, *Null*).

5) *temp* = len (*additional_input*).

6) If (*temp* > *max_length*), then Return (Failure message, *Null*).

7) If (*requested_no_of_bits* > *max_request_length*), then Return (Failure message, *Null*).

8) If ((*prediction_resistance_request_flag* = Provide_prediction_resistance) and (*prediction_resistance_flag* = No_prediction_resistance)), then Return (Failure message, *Null*).

NOTE 1 If an implementation does not need the *prediction_resistance_flag*, then the *prediction_resistance_flag* can be omitted as an input parameter, and step 8 can be omitted.

9) If ((*reseed_counter* > *reseed_interval*) OR (*prediction_resistance_request_flag* = Provide_prediction_resistance)), then:

   9.1 If reseeding is not available, then Return (Failure message, *Null*).

   9.2 *status* = Reseed_HMAC_DRBG_Instantiation (*state_handle*, *additional_input*).

```
    NOTE 2 If an implementation never provides additional_input, then a Null string replaces the
    additional_input in step 9.2.

    9.3  If (status ≠ "Success"), then Return (Failure message, Null).

    9.4  V = state(state_handle).V, Key = state(state_handle).Key, reseed_counter = state(state_handle).
         reseed_counter.

    9.5  additional_input = Null.

10) If (additional_input ≠ Null), then:

    10.1 (Key, V) = HMAC_DRBG_Update(additional_input, Key, V).

11) temp = Null.

12) While (len (temp) < requested_number_of_bits) do:

    12.1     V = HMAC (Key, V).

    12.2     temp = temp || V.

13) pseudorandom_bits = Leftmost requested_no_of_bits of temp.

14) (Key, V) = HMAC_DRBG_Update(additional_input, Key, V).

15) reseed_counter = reseed_counter + 1.

16) Update the changed values in the state.

    16.1state(state_handle).Key = Key.

    16.2     state(state_handle).V = V.

    16.3     state(state_handle).reseed_counter = reseed_counter.

17) Return ("Success", pseudorandom_bits).
```

C.3 DRBGs based on block ciphers

C.3.1 Introduction to DRBGs based on block ciphers

A block cipher DRBG is based on a block cipher algorithm.

The block cipher DRBGs specified in this document have been designed to use any block cipher algorithm specified by ISO/IEC 18033-3 and ISO/IEC 29192-2 and may be used by applications requiring various security strengths. The following are provided as DRBGs based on block cipher algorithms:

1) The CTR_DRBG (...) specified in C.3.2.

2) The OFB_DRBG (...) specified in C.3.3.

Table C.2 provides an example of a block cipher algorithm specified by ISO/IEC 18033-3 and ISO/IEC 29192-2, illustrating the security strengths, entropy and seed requirements that are used.

Table C.2 — Security strengths, entropy and seed length requirements for the AES-128, 192 and 256 block cipher

Block cipher algorithm	Security strengths	Required minimum entropy	Seed length (in bits)
AES-128	112, 128	128	256
AES-192	112, 128, 192	192	320
AES-256	112, 128, 192, 256	256	384

C.3.2 CTR_DRBG

C.3.2.1 Discussion

The CTR_DRBG (...) uses a block cipher algorithm in counter mode. The block cipher algorithm and the counter mode are as specified in ISO/IEC 18033-3 and ISO/IEC 29192-2, and ISO/IEC 10116, respectively. The same block cipher algorithm and key length are used for all block cipher operations.

C.3.2.2 Description

C.3.2.2.1 General

The instantiation and reseeding of CTR_DRBG (...) consists of obtaining a *seed value* with the appropriate amount of entropy. The seed value is used to derive a *seed*, which is then used to derive elements of the initial *state* of the DRBG. The *state* consists of:

1) the value *V*, which is updated each time another *outlen-bits* of output are produced (where *outlen* is the number of output bits from the underlying block cipher algorithm);

2) the *Key*, which is updated whenever a predetermined number of output blocks are generated;

3) the key length (*keylen*) to be used by the block cipher algorithm;

4) the security *strength* of the DRBG instantiation;

5) a counter (*reseed_counter*) that indicates the number of requests for pseudorandom bits since instantiation or reseeding; and

6) a *prediction_resistance_flag* that indicates whether or not a prediction resistance capability is required for the DRBG.

The variables used in the description of CTR_DRBG (...) are:

additional_input Optional additional input, which is ≤ *max_length* bits in length.

Block_Cipher (*Key*, *V*) The block cipher algorithm, where *Key* is the key to be used, and *V* is the input block.

Block_Cipher_df (*a*, *b*) The block cipher derivation function.

outlen The length of the block cipher algorithm's output block.

ctrlen Counter field length, with $4 \le ctrlen \le blocklen$.

seed_value The bits containing entropy that are used to determine the *seed_material* and generate a *seed*.

Get_entropy (*min_entropy*, *min_length*, *max_length*)
A function that acquires a string of bits from a randomness source. *min_entropy* indicates the minimum amount of entropy to be provided in the returned bits;
min_length indicates the minimum number of bits to be returned;
max_length indicates the maximum number of bits to be returned.

Key
The key used to generate pseudorandom bits.

keylen
The length of the key for the block cipher algorithm.

len (*x*)
A function that returns the number of bits in input string *x*.

max_length
The maximum length of a string for obtaining entropy. When a derivation function is used, this value is implementation dependent but is ≤ 2^{35} bits (for a 128-bit block cipher, there is a maximum of 2^{28} blocks, i.e. 2^{32} bytes – 2^{35} bits). When a derivation function is not used, then *max_length* = *seedlen*.

max_request_length
The maximum number of pseudorandom bits that may be requested during a single request; this value is implementation dependent but is ≤ 2^{35} bits for 128-bit block ciphers, and 64-bit block ciphers are no longer approved.

min_entropy
The minimum amount of entropy to be obtained from the entropy source and provided in the *seed value*.

Null
The null (i.e. empty) string.

personalization_string
A personalization string.

prediction_resistance_flag
A flag indicating whether or not enhanced forward secrecy requests should be handled:

prediction_resistance_flag = 1 = allow = Allow_prediction_resistance

prediction_resistance_flag = 0 = do not allow =

No_prediction_resistance.

prediction_resistance_request_flag
Indicates whether or not enhanced forward secrecy is required during a request for pseudorandom bits:

prediction_resistance_request_flag = 1 =

Provide_prediction_resistance

prediction_resistance_request_flag = 0 = No_prediction_resistance.

pseudorandom_bits
The pseudorandom bits produced during a single call to the CTR_DRBG (...) process.

requested_no_of_bits
The number of pseudorandom bits to be returned from CTR_DRBG (...) function.

requested_strength
The security strength to be associated with the requested pseudorandom bits.

reseed_counter
A count of the number of requests for pseudorandom bits since the instantiation or reseeding.

reseed_interval
The maximum number of requests for the generation of pseudorandom bits before reseeding is required. The maximum value is ≤ 2^{32} for 128-bit block ciphers, and ≤ 2^{16} for 64-bit block ciphers.

seedlen The length of the seed, where *seedlen* = *outlen* + *keylen*.

seed_material The data used as the *seed*.

state (*state_handle*) An array of states for different DRBG instantiations. A *state* is carried between DRBG calls. For the CTR_DRBG (...), the *state* for an instantiation is defined as *state* (*state_handle*) = {*V*, *Key*, *keylen*, *strength*, *reseed_counter*, *prediction_resistance_flag*}. A particular element of the *state* is specified as *state*(*state_handle*).*element*, e.g. *state* (*state_handle*).*V*.

state_handle A handle to the state space for the given instantiation.

status The status returned from a function call, where *status* = “Success” or a failure message.

strength The security strength provided by the DRBG instantiation.

temp A temporary value.

V A value in the *state* that is updated whenever pseudorandom bits are generated.

C.3.2.2.2 Instantiation of CTR_DRBG(...)

The following process or its equivalent is used to initially instantiate the CTR_DRBG (...) process.

Instantiate_CTR_DRBG (...):

```
Input: integer (requested_strength, prediction_resistance_flag), string personalization_string.

Output: string status, integer state_handle.

Process:

    1) If (requested_strength > the maximum security strength that can be provided for the block cipher
       algorithm) then Return (Failure message).

    2) If (requested_strength  112), then (strength = 112; keylen = 128)

                  Else if (requested_strength  128), then (strength = 128; keylen = 128)

                      Else if (requested_strength  192), then (strength = 192; keylen = 192)

                          Else (strength = 256; keylen = 256).

    NOTE 1 This step sets the strength to one of the four security strengths and determines the key length.

    3) seedlen = outlen + keylen.

    4) temp = len (personalization_string).

    5) If (temp > max_length), then Return (Failure message).

    NOTE 2 If a personalization_string is never provided, then the personalization_string input parameter
    and steps 4 and 5 can be omitted.

    6) The following code is used when a derivation function is available (it is possible that a source of
       full entropy is not available).

        6.1 min_entropy = strength + |nonce|.
        6.2  (status, seed_value) = Get_entropy (min_entropy, min_entropy, max_length).

        6.3  If (status ≠ "Success"), then Return (Failure message).

        6.4  seed_material = seed_value || nonce || personalization_string.

        NOTE 3  If no personalization_string is provided in step 6.4, then step 6.4 can be omitted, and
        step 6.5 becomes seed_material = Block_Cipher_df (seed_value, seedlen).

        6.5  seed_material = Block_Cipher_df (seed_material, seedlen).

    7)  The following code is used when a full entropy source is known to be available and a derivation
        function is not used.

        7.1  (status, entropy_input) = Get_entropy (seedlen, seedlen, seedlen).

        7.2  If (status ≠ "Success"), then Return (Failure message).

        7.3  If (temp < seedlen), then personalization_string = personalization_string || $0^{seedlen - temp}$.

        NOTE 4  If the personalization_string is too short, it is padded with zeroes.

        7.4  seed_material = entropy_input $\oplus$ personalization_string.

        NOTE 5  If no personalization_string is provided above, then steps 7.3 and 7.4 are omitted, and
        step 7.1 becomes: (status, seed_material) = Get_entropy (seedlen, seedlen, seedlen).

    8)  Key = $0^{keylen}$.

    9)  V = $0^{outlen}$.

    10) (Key, V) = Update (seed_material, keylen, Key, V).

    11) reseed_counter = 1.

    12) state (state_handle) = {V, Key, keylen, strength, reseed_counter, prediction_resistance_flag}.

    NOTE 6  If an implementation does not require the prediction_resistance_flag as a calling parameter
    [i.e. the CTR_DRBG (...) routine in C.3.2.2.7 either always or never acquires new entropy in step 9], then
    the prediction_resistance_flag in the calling parameters and in the state (see step 12) can be omitted.

    13) Return ("Success", state_handle).
```

C.3.2.2.3 Internal function: the Update function

The Update (...) function updates the internal state of the CTR_DRBG (...) using *seed_material*, which is *seedlen-bits* in length. The following or an equivalent process is used as the Update (...) function.

Update (...):

```
  Input: integer keylen, bit-string (seed_material, Key, V).

  Output: bit-string (Key, V).

  Process:

    1)  seedlen = outlen + keylen.

    2)  temp = Null.

    3)  While (len (temp) < seedlen) do:
        3.1  $V = (V + 1) \bmod 2^{outlen}$.
        3.2  $output\_block$ = Block_Cipher ($Key$, $V$).
        3.3  $temp = temp \,||\, output\_block$.
    4)  $temp$ = Leftmost $seedlen$ bits of $temp$.
    5)  $temp = temp \oplus seed\_material$.
    6)  $Key$ = Leftmost $keylen$ bits of $temp$.
    7)  $V$ = Rightmost $blocklen$ bits of $temp$.
    8)  Return ($Key$, $V$).
```

C.3.2.2.4 Derivation function using a block cipher algorithm

Let CBC_MAC be the function as specified at C.3.2.2.5. Let Block_Cipher be an encryption operation in ECB mode using the selected block cipher algorithm. Let *outlen* be its output block, let *keylen* be the key length.

Input:

1) bit-string *input_string*: The string to be operated on.

2) integer *no_of_bits*: the number of bits returned by Block_Cipher_df.

Output: bit-string *requested_bits*: the result of performing the Block_Cipher_df.

Process:

```
    1)  $L$ = len ($input\_string$) / 8.

    NOTE 1   $L$ is the bit-string representation of the integer resulting from len($input\_string$)/8. $L$ is represented as a 32-bit integer.

    2)  $N = no\_of\_bits$ / 8.

    NOTE 2   $N$ is the bit-string representation of the integer resulting from $no\_of\_bits$ /8. $N$ is represented as a 32-bit integer.

    3)  $S = L \,||\, N \,||\, input\_string \,||$ 0x80.

    NOTE 3   This step prepends the string length and the requested length of the output to the $input\_string$.

    4)  If necessary, pad $S$ with zeroes.
    5)  While (len ($S$) mod $outlen \neq 0$) $S = S \,||$ 0x00.
    6)  $temp$ = the $Null$ string.
    7)  $i = 0$.

    NOTE 4   $i$ is represented as a 32-bit integer, i.e., len( $i$ ) = 32.

    8)  $K$ = Leftmost $keylen$ bits of 0x00010203 … 1F.
    9)  While (len ($temp$) < $keylen + outlen$) do:
        a)  $IV = i \,||\, 0^{outlen - \mathrm{len}(i)}$ .

        NOTE 5   This step pads the integer representation of $i$ and is padded with zeroes to $outlen$ bits.
        b) $temp$ = $temp$ || CBC_MAC ($K$,($IV$ || $S$)).
        c) $i$ =$i$ + 1)
    10) $K$ = Leftmost $keylen$ bits of $temp$.
    11) $X$ = Next $outlen$ bits of $temp$.
    12) $temp$ = the $Null$ string.
    13) While (len ($temp$) < $no\_of\_bits$) do:
        a) $X$ = Block_Cipher ($K$, $X$).
        b) $temp$ = $temp$ || $X$.
    14) $requested\_bits$ = Leftmost $no\_of\_bits$ bits of $temp$.
    15) Return ($requested\_bits$).
```

C.3.2.2.5 CBC_MAC function

The CBC_MAC function is a method for computing a message authentication code. Let Block_Cipher be an encryption operation in the ECB mode using the selected block cipher algorithm. Let *outlen* be the length of the output block of the block cipher algorithm to be used.

The following or an equivalent process is used to derive the requested number of bits.

Input:

1) bit-string *Key*: The key to be used for the block cipher operation.

2) bit-string *data_to_MAC*: The data to be operated upon.

Output: bit-string *output_block*: The result to be returned from the CBC_MAC operation.

Process:

```
    1) $chaining\_value$ =$0^{outlen}$ .
    NOTE   This step sets the first chaining value to $outlen$ zeroes.
    2) $n$ = len ($data\_to\_MAC$) / $outlen$.
    3) Split the $data\_to\_MAC$ into $n$ blocks of $outlen$-$bits$ each forming $block_1$ to $block_n$.
    4) For $i$ = 1 to $n$ do:
        4.1 input_block = chaining_value $\oplus$block$_i$.
        4.2 chaining_value= Block_Cipher (Key, input_block).
    5) $output\_block$ = $chaining\_value$.
    6) Return $output\_block$.
```

C.3.2.2.6 Reseeding CTR_DRBG(...) Instantiation

The following or an equivalent process is used to explicitly reseed the CTR_DRBG (...) process.

Reseed_CTR_DRBG_Instantiation (...):

Input: integer *state_handle*, string *additional_input*.

Output: string *status*.

Process:

1) If the state indicated by *state_handle* is not available, then Return (Failure message).

2) Get the appropriate *state* values, e.g., *V* = *state*(*state_handle*).*V*, *Key* = *state*(*state_handle*).*Key*, *keylen* = *state*(*state_handle*).*keylen*, *strength* = *state*(*state_handle*).*strength*, *prediction_resistance_flag* = *state*(*state_handle*).*prediction_resistance_flag*.

3) *seedlen* = *outlen* + *keylen*.

4) *temp* = len (*additional_input*).

NOTE 1 If an implementation does not handle *additional_input*, then the *additional_input* parameter of the input can be omitted as well as steps 4 and 5 below.

5) If (*temp* > *max_length*), then Return (Failure message).

6) The following code is used when a derivation function is available (a source of full entropy may or may not be available).

6.1 *min_entropy* = *strength* + 64.

6.2 (*status*, *seed_value*) = Get_entropy (*min_entropy*, *min_entropy*, *max_length*).

6.3 If (*status* ≠ “Success”), then Return (Failure message).

6.4 *seed_material* = *seed_value* || *additional_input*.

NOTE 2 If an implementation does not handle *additional_input*, then step 6.4 can be omitted, and step 6.5 can be changed to: *seed_material* = Block_Cipher_df (*seed_value*, *seedlen*).

6.5 *seed_material* = Block_Cipher_df (seed_material, seedlen).

7) The following code is used when a full entropy source is known to be available and a derivation function is not used.

7.1 (*status*, *entropy_input*) = Get_entropy (*seedlen*, *seedlen*, *seedlen*).

7.2 If (*status* ≠ “Success”), then Return (Failure message).

7.3 If (*temp* < *seedlen*), then *additional_input* = *additional_input* || $0^{seedlen\ -\ temp}$.

NOTE 3 This step pads with zeroes if the *additional_input* is too short.

7.4 *seed_material* = *seed_value* ⊕ *additional_input*.

NOTE 4 If an implementation does not handle *additional_input*, then steps 7.3 and 7.4 can be omitted, and step 7.1 can be changed to: (*status*, *seed_material*) = Get_entropy (*seedlen*, *seedlen*, *seedlen*).

8) (*Key*, *V*) = Update (*seed_material*, *keylen*, *Key*, *V*).

9) *reseed_counter* = 1.

10) *state*(*state_handle*) = {*V*, *Key*, *keylen*, *strength*, *reseed_counter*, *prediction_resistance_flag* }.

11) Return (“Success”).

C.3.2.2.7 Generating pseudorandom bits using CTR_DRBG (...)

The following process or an equivalent is used to generate pseudorandom bits.

CTR_DRBG (...):

Input: integer (*state_handle*, *requested_no_of_bits*, *requested_strength*, *prediction_resistance_request_flag*), string *additional_input*.

Output: string *status*, bit-string *pseudorandom_bits*.

Process:

1) If a state is not available, then Return (Failure message, *Null*).

2) Get the appropriate *state* values, e.g. *V* = *state*(*state_handle*).*V*, *Key* = *state*(*state_handle*).*Key*, *keylen* = *state*(*state_handle*).*keylen*, *strength* = *state*(*state_handle*).*strength*, *reseed_counter* = *state*(*state_handle*).*reseed_counter*, *prediction_resistance_flag* = *state*(*state_handle*).*prediction_resistance_flag*.

3) If (*requested_strength* > *strength*), then Return (Failure message, *Null*).

4) *seedlen* = *outlen* + *keylen*.

5) *temp* = len (*additional_input*).

NOTE 1 If an implementation never provides *additional_input*, then the *additional_input* input parameter and steps 5 and 6 can be omitted.

6) If (*temp* > *max_length*), then Return (Failure message, *Null*).

7) If (*requested_no_of_bits* > *max_request_length*), then Return (Failure message, *Null*).

8) If ((*prediction_resistance_request_flag* = Provide_prediction_resistance) and (*prediction_resistance_flag* = No_prediction_resistance)), then Return (Failure message, *Null*).

NOTE 2 If an implementation does not need the *prediction_resistance_flag*, then the *prediction_resistance_flag* can be omitted as an input parameter and step 8 can be omitted.

9) If ((*reseed_counter* > *reseed_interval*) OR (*prediction_resistance_request_flag* = Provide_prediction_resistance)), then:

   Return (Failure message, *Null*).

   NOTE 3 This is the case if reseeding is not available.

   9.1 *status* = Reseed_CTR_DRBG_Instantiation (*state_handle*, *additional_input*).

   NOTE 4 If an implementation will never provide *additional_input*, then a *Null* string replaces the *additional_input* in step 9.1)

   9.2 If (*status* ≠ “Success”), then Return (Failure message, *Null*).

   9.3 *V* = *state*(*state_handle*).*V*, *Key* = *state*(*state_handle*).*Key*, *reseed_counter* = *state*(*state_handle*).*reseed_counter*.

   9.4 *additional_input* = *Null*.

10) If (*additional_input* = *Null*), then *additional_input* = $0^{seedlen}$.

NOTE 5 If an implementation never provides *additional_input*, then step 10 can be omitted.

11) The following code is used when a derivation function is available (a source of full entropy is possibly not available).

If (*additional_input* ≠ *Null*), then:

11.1 *additional_input* = Block_Cipher_df (additional_input, seedlen).

NOTE 6 Derive *seedlen* bits.

11.2 (Key, V) = Update (additional_input, keylen, Key, V).

NOTE 7 If an implementation never provides *additional_input*, then step 11 can be omitted.

12) The following code is used when a full entropy source is known to be available and a derivation function is not used.

If (*additional_input* ≠ *Null*), then:

12.1 temp = len (additional_input).

12.2 If (temp < seedlen), then additional_input = additional_input || $0^{\text{seedlen - temp}}$.

NOTE 8 If the length of the *additional_input* is < *seedlen*, pad with zeroes to *seedlen* bits.

12.3 (Key, V) = Update (additional_input, keylen, Key, V).

NOTE 9 If an implementation never provides *additional_input*, then step 12 can be omitted.

13) *temp* = *Null*.

14) While (len (*temp*) < *requested_no_of_bits*) do:

14.1 If $ctrlen < blocklen$

14.1.1 $inc = Rightmost(V, ctrlen) + 1 \bmod 2^{ctrlen}$

14.1.2 $V = Leftmost(V, blocklen - ctrlen) \| inc$

Else $V = (V + 1) \bmod 2^{ctrlen}$

14.2 *output_block* = Block_Cipher (*Key*, *V*).

14.3 temp = temp || output_block.

15) *pseudorandom_bits* = Leftmost *requested_no_of_bits* bits of *temp*.

16) (*Key*, *V*) = Update (*additional_input*, *keylen*, *Key*, *V*).

NOTE 10 Update performed for backward secrecy.

NOTE 11 If an implementation never provides *additional_input*, then step 16 becomes (*Key*, *V*) = Update ($0^{seedlen}$, *keylen*, *Key*, *V*).

17) *reseed_counter* = *reseed_counter* + 1.

18) *state*(*state_handle*) = {*V*, *Key*, *keylen*, *strength*, *reseed_counter*, *prediction_resistance_flag*}.

19) Return (“Success”, *pseudorandom_bits*).

C.3.3 OFB_DRBG

C.3.3.1 Discussion

OFB_DRBG (...) uses a block cipher algorithm specified by ISO/IEC 18033-3 and ISO/IEC 29192-2 in the output feedback mode specified by ISO/IEC 10116. The same block cipher algorithm and key length are used for all block cipher operations. The block cipher algorithm and key size meet or exceed the security requirements of the consuming application.

C.3.3.2 Description

C.3.3.2.1 General

The instantiation and reseeding of OFB_DRBG (...) consists of obtaining a *seed value* with the appropriate amount of entropy. The seed value is used to derive a *seed*, which is then used to derive elements of the initial *state* of the DRBG. The *state* consists of:

1) the value *V*, which is updated each time another *outlen-bits* of output are produced (where *outlen* is the number of output bits from the underlying block cipher algorithm);

2) the *Key*, which is updated whenever a predetermined number of output blocks are generated;

3) the key length (*keylen*) to be used by the block cipher algorithm;

4) the security *strength* of the DRBG instantiation;

5) a counter (*reseed_counter*) that indicates the number of requests for pseudorandom bits since instantiation or reseeding; and

6) a *prediction_resistance_flag* that indicates whether or not a prediction resistance capability is required for the DRBG.

The variables for OFB_DRBG (...) are the same as those used for the CTR_DRBG (...) specified in C.3.2.2.1.

C.3.3.2.2 Internal function: The Update function

The Update (...) function updates the internal state of the OFB_DRBG (...) using *seed_material*, which is *seedlen-bits* in length. The following or an equivalent process is used as the Update (...) function.

```
Update (...):
    Input: integer keylen, bit-string (seed_material, Key, V).
    Output: bit-string (Key, V).
    Process:
        1) $seedlen = outlen + keylen$.
        2) $temp = Null$.
        3) While (len ($temp$) < $seedlen$) do:
            3.1 $V$ = Block_Cipher ($Key$, $V$).
            3.2 $temp = temp \| V$.
        4) $temp$ = Leftmost $seedlen$ bits of $temp$.
        5) $temp = temp \oplus seed\_material$.
```

6) *Key* = Leftmost *keylen* bits of *temp*.

7) *V* = Rightmost *blocklen* bits of *temp*.

8) Return (*Key*, *V*).

NOTE The only difference between the update function for OFB_DRBG (...) and CTR_DRBG (...) is in step 3.

C.3.3.2.3 Instantiation of OFB_DRBG (...)

This process is the same as the instantiation process for CTR_DRBG (...) in C.3.2.2.2.

C.3.3.2.4 Reseeding OFB_DRBG (...) Instantiation

This process is the same as the reseeding process for CTR_DRBG (...) in C.3.2.2.6.

C.3.3.2.5 Generating pseudorandom bits using OFB_DRBG (...)

OFB_DRBG (...):

Input: integer (*state_handle*, *requested_no_of_bits*, *requested_strength*, *prediction_resistance_request_flag*), string *additional_input*.

Output: string *status*, bit-string *pseudorandom_bits*.

Process:

1) If a state is not available, then Return (Failure message, *Null*).

2) Get the appropriate *state* values, e.g. *V* = *state*(*state_handle*).*V*, *Key* = *state*(*state_handle*).*Key*, *keylen* = *state*(*state_handle*).*keylen*, *strength* = *state*(*state_handle*).*strength*, *reseed_counter* = *state*(*state_handle*).*reseed_counter*, *prediction_resistance_flag* = *state*(*state_handle*).*prediction_resistance_flag*.

3) If (*requested_strength* > *strength*), then Return (Failure message, *Null*).

4) *seedlen* = *outlen* + *keylen*.

5) *temp* = len (*additional_input*).

NOTE 1 If an implementation never provides *additional_input*, then the *additional_input* input parameter and steps 5 and 6 can be omitted.

6) If (*temp* > *max_length*), then Return (Failure message, *Null*).

7) If (*requested_no_of_bits* > *max_request_length*), then Return (Failure message, *Null*).

8) If ((*prediction_resistance_request_flag* = Provide_prediction_resistance) and (*prediction_resistance_flag* = No_prediction_resistance)), then Return (Failure message, *Null*).

NOTE 2 If an implementation does not require the *prediction_resistance_flag*, then the *prediction_resistance_flag* can be omitted as an input parameter and step 8 can be omitted.

9) If ((*reseed_counter* > *reseed_interval*) OR (*prediction_resistance_request_flag* = Provide_prediction_resistance)), then:

Return (Failure message, *Null*).

NOTE 3 This is the case if reseeding is not available.

9.1 *status* = Reseed_CTR_DRBG_Instantiation (*state_handle*, *additional_input*).

NOTE 4 If an implementation never provides *additional_input*, then a *Null* string replaces the *additional_input* in step 9.1.

9.2 If (*status* ≠ “Success”), then Return (Failure message, *Null*).

9.3 *V* = *state*(*state_handle*).*V*, *Key* = *state*(*state_handle*).*Key*, *reseed_counter* = *state*(*state_handle*).*reseed_counter*.

9.4 *additional_input* = *Null*.

10) If (*additional_input* = *Null*), then *additional_input* = $0^{seedlen}$.

NOTE 5 If an implementation will never provide *additional_input*, then step 10 can be omitted.

11) The following code is used when a derivation function is available (it is possible that a source of full entropy is not available.

If (*additional_input* ≠ *Null*), then:

11.1 *additional_input* = Block_Cipher_df (*additional_input*, *seedlen*).

NOTE 6 Derived *seedlen* bits.

11.2 (*Key*, *V*) = Update (*additional_input*, *keylen*, *Key*, *V*).

NOTE 7 If an implementation never provides *additional_input*, then step 11 can be omitted.

12) The following code is used when a full entropy source is known to be available and a derivation function is not used.

If (*additional_input* ≠ *Null*), then:

a) *temp* = len (*additional_input*).

b) If (*temp* < *seedlen*), then *additional_input* = *additional_input* || $0^{seedlen\ -\ temp}$.

c) If the length of the additional_input is < *seedlen*, pad with zeroes to *seedlen-bits*.

d) (*Key*, *V*) = Update (*additional_input*, *keylen*, *Key*, *V*).

NOTE 8 If an implementation will never provide *additional_input*, then step 12 can be omitted.

13) *temp* = *Null*.

14) While (len (*temp*) < *requested_no_of_bits*) do:

14.1) *V* = Block_Cipher (*Key*, *V*).

14.2) *temp* = *temp* || *V*.

15) *pseudorandom_bits* = Leftmost *requested_no_of_bits* bits of *temp*.

16) (*Key*, *V*) = Update (*additional_input*, *keylen*, *Key*, *V*).

NOTE 9 Update performed for backward secrecy.

NOTE 10 If an implementation never provides *additional_input*, then step 16 becomes (*Key*, *V*) = Update ($0^{seedlen}$, *keylen*, *Key*, *V*).

17) *reseed_counter* = *reseed_counter* + 1.

18) *state*(*state_handle*) = {*V*, *Key*, *keylen*, *strength*, *reseed_counter*, *prediction_resistance_flag*}.

19) Return (“Success”, *pseudorandom_bits*).

NOTE 11 The only difference between OFB_DRBG (...) and CTR_DRBG (...) in C.3.2.2.7 is in step 14.

Annex D
(informative)

NRBG examples

D.1 Canonical coin tossing example

D.1.1 Overview

This non-deterministic random bit generator (NRBG) example is illustrative of some of the requirements that go into the design of an NRBG.

For some applications, it is necessary to non-deterministically generate random bits without using a full-featured NRBG. Coin tossing is commonly considered to be a standard intuitive model for random bit generation, and, under fairly simple assumptions, the addition of some mathematically based post-processing of the coin toss outcomes results in perfectly independent and unbiased binary outputs. This document permits the generation of random bits using a coin tossing procedure. Such a procedure can be impractical for an application requiring large numbers of random bits; however, if the application accepts expansion by a DRBG, coin-tossing can provide the relatively short seeds required by the DRBG.

This document recommends an NRBG that requires the inclusion of various components and functions that effectively amount to a DRBG operating in conjunction with one or more randomness sources. These components are designed to result in a process that not only generates essentially unbiased and independent bits but also tolerates various harmful situations through the inclusion of safety margins and interactions among the components. However, the canonical coin flip RBG described here does not directly follow this approach. It is possible to argue that in situations where a coin tossing NRBG is appropriate, the objectives to be met by these NRBG components are either obviously unnecessary because of the environment or are satisfied via certain procedural steps in the coin tossing process. The coin tossing procedure described in this annex is a canonical NRBG. A process for an NRBG based on coin tossing is described in D.1.2.

D.1.2 Description of basic process

The basic coin tossing NRBG permitted by this document consists of a person repeatedly flipping a single coin, assigning one side of the coin to the outcome “zero” and the other side of the coin to the outcome “one,” and performing the Peres Unbiasing procedure[25] on the resulting output sequence (see D.1.5 for a detailed description of the Peres Unbiasing procedure). Assuming that the coin toss outcomes are independent and identically distributed (but not necessarily unbiased), the Peres Unbiasing procedure is known to produce unbiased output, with the number of output bits produced asymptotically equal to the Shannon entropy in the coin toss sequence.

D.1.3 Relation to standard NRBG components

The canonical coin tossing NRBG is acceptable as an NRBG. The correspondence between the features of this canonical NRBG and the features of the NRBG functional model is as follows.

1) Primary entropy source: A coin. It is important to note that owing to the Peres Unbiasing procedure, the coin is not required to be perfectly unbiased, although it is assumed that any bias is stationary.

2) Additional inputs: One or more additional coins used for increased assurance or trust among multiple parties.

3) Internal state: The values in each of the registers and variables used to implement the Peres Unbiasing procedure.