INTERNATIONAL STANDARD

ISO/IEC 18031

Second edition
2011-11-15

Information technology — Security techniques — Random bit generation

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

Reference number
ISO/IEC 18031:2011(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web 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. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 18031 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 27, *IT Security techniques*.

This second edition cancels and replaces the first edition (ISO/IEC 18031:2005), which has been technically revised. It also incorporates the Technical Corrigendum ISO/IEC 18031:2005/Cor.1:2009.

Introduction

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

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

- random keys and initialisation 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 PIN and password generation;
- nonces.

The purpose of this International Standard 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. This International Standard identifies the two types as non-deterministic and deterministic random bit generators.

A non-deterministic random bit generator can be defined as a random bit generating mechanism that 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, may or may not be publicly available. The seed is processed by a function which provides an output.

NOTE This International Standard also recognizes and discusses the existence of Hybrid Random Bit Generators.

Information technology — Security techniques — Random bit generation

1 Scope

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

This International Standard

- specifies the characteristics of the main elements required for a non-deterministic random bit generator,
- specifies the characteristics of the main elements required for a deterministic random bit generator,
- establishes the security requirements for both the non-deterministic and the deterministic random bit generator.

Where there is a requirement to produce sequences of random numbers from random bit strings, Annex B gives guidelines on how this can be performed.

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 International Standard.

2 Normative references

The following referenced documents are indispensable for the application 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 technology — Security techniques — Message Authentication Codes (MACs) — Part 2: Mechanisms using a dedicated hash-function*

ISO/IEC 10116, *Information technology — Security techniques — Modes of operation for an n-bit block cipher*

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

ISO/IEC 18032, *Information technology — Security techniques — Prime number generation*

ISO/IEC 18033-3, *Information technology — Security techniques — Encryption algorithms — Part 3: Block ciphers*

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

3 Terms and definitions

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

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 values cannot be determined from knowledge of the current value or subsequent values

3.3
biased source
source of bit strings (or numbers) from a sample space where some bit strings (or numbers) are more likely to be chosen than some other bit strings (or numbers)

NOTE 1 Equivalently, if the sample space consists of r elements, some elements will occur with probability different from $1/r$.

NOTE 2 This term can be contrasted with unbiased source (3.35).

3.4
bit stream
continuous output of bits from a device or mechanism

3.5
bit string
finite sequence of ones and zeroes

3.6
black box
idealized mechanism that accepts inputs and produces outputs, but is designed such that an observer cannot see inside the box or determine exactly what is happening inside that box

NOTE This term can be contrasted with glass box (3.14).

3.7
block cipher
symmetric encipherment system with the property that the encryption operates on a block of plaintext, i.e. a string of bits of a defined length, to yield a block of ciphertext

[ISO/IEC 18033-1]

3.8
cryptographic boundary
explicitly defined continuous perimeter that establishes the physical bounds of a cryptographic module and contains all the hardware, software and/or firmware components of a cryptographic module

[ISO/IEC 19790]

3.9
deterministic algorithm
characteristic of an algorithm that states that given the same input, the same output is always produced

3.10
deterministic random bit generator
DRBG
random bit generator that produces a random-appearing sequence of bits by applying a deterministic algorithm 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 In particular, non-deterministic sources may also form part of these secondary inputs.

3.11
entropy
measure of the disorder, randomness or variability in a closed system

NOTE The entropy of a random variable X is a mathematical measure of the amount of information provided by an observation of X.

3.12
entropy source
component, device or event which produces outputs which, when captured and processed in some way, produce a bit string containing entropy

3.13
forward secrecy
assurance that the knowledge of subsequent (future) values cannot be determined from current or previous values

3.14
glass box
idealized mechanism that accepts inputs and produces outputs and is designed such that an observer can see inside and determine exactly what is going on

NOTE This term can be contrasted with black box (3.6).

3.15
hash-function
function which maps strings of bits to fixed-length strings of bits, satisfying the following two properties.

- It is computationally infeasible to find for a given output, an input that maps to this output.
- It is computationally infeasible to find for a given input, a second input, which maps to the same output

NOTE Computational feasibility depends on the specific security requirements and environment.

[ISO/IEC 10118-1]

3.16
human entropy source
entropy source that has some kind of random human component

3.17
hybrid DRBG
DRBG that uses a non-deterministic entropy source as an additional entropy source

3.18
hybrid NRBG
NRBG that takes a seed value as an additional entropy source

NOTE Hybrid NRBG may be physical or non-physical.

3.19
initialisation value
value used in defining the starting point of a cryptographic algorithm

NOTE An example of where an initialisation value is used is in a hash-function or an encryption algorithm.

3.20
Kerckhoffs 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 is not known to an adversary

NOTE A Kerckhoffs box lies between a black box and a glass box in terms of the knowledge of an adversary.

3.21
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 Known-answer testing of a deterministic mechanism may also include testing the integrity of the software which 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.

3.22
min-entropy
lower bound of entropy that is useful in determining a worst-case estimate of sampled entropy

NOTE 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.

3.23
non-deterministic random bit generator
NRBG
RBG whose security depends upon sampling an entropy source

NOTE The entropy source will be sampled whenever the RBG produces output, and possibly more often.

3.24
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 for a given output an input which maps to this output

[ISO/IEC 11770-3]

3.25
output generation function
function in an RBG that computes the output of the RBG from the internal state of the RBG

3.26
pseudorandom sequence of bits
sequence of bits or a number that appears to be selected at random even though the selection process is done by a deterministic algorithm

3.27
pure DRBG
DRBG whose entropy sources are seeds

3.28
pure NRBG
NRBG whose entropy sources are non-deterministic

NOTE The pure NRBG may be physical or non-physical.

3.29
random bit generator
RBG
device or algorithm that outputs a sequence of bits that appears to be statistically independent and unbiased

3.30
reseeding
specialised internal state transition function which updates the internal state in the event that a new seed value is supplied

NOTE The usage of the term 'reseeding' is not unique in the literature. Some authors denote as 'reseeding' each mechanism that replaces the current value of the internal state by a fresh value. This International Standard follows this terminology. However, often one distinguishes between 'reseeding' and 'seed update'. The term 'reseeding' then only comprises mechanisms that replace the internal state by a new value, which does not depend on the current value (essentially a new seeding process). In contrast 'seed update' denotes a mechanism that computes the new internal state from its current value and other (usually non-deterministic) data (cf. 9.6, item 3).

3.31
secret parameter
input to the RBG during initialisation, which provides additional entropy in the case of an entropy source failure or compromise

3.32
seed
string of bits that is used as input to a DRBG

NOTE The seed will determine a portion of the state of the DRBG.

3.33
seedlife
period of time between initialising the DRBG with one seed and reseeding (fully initialising) that DRBG with another seed

3.34
state
condition of a random bit generator or any part thereof with respect to time and circumstance

3.35
unbiased source
source of bit strings (or numbers) from a sample space where all potential bit strings (or numbers) have the same possibility of being chosen

NOTE 1 Equivalently, if the sample space consists of r elements, all elements will occur with probability $1/r$.

NOTE 2 This term can be contrasted with biased source (3.3).

4 Symbols

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

Symbol	Meaning
Pr[x]	Probability of occurrence of x.
IV	Initialisation 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 bit wise addition mod 2) of two bit strings X and Y of the same length.
$X \| \| Y$	Concatenation of two bit strings X and Y in that order.
$\| a \|$	The length in bits of string a.
x mod n	The unique remainder r, $0 \leq r \leq n$-1, when integer x is divided by n. For example, 23 mod 7 = 2.
	Used in a figure to illustrate a “switch” between sources of input.

5 Properties and requirements of an RBG

5.1 Properties of an RBG

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

1. 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 International Standard;

2. Unbiased: That is, each potential outcome has the same chance of occurring; and

3. 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 an RBG. The RBGs specified in this International Standard will try to simulate a series of idealized coin flips.

As indicated above, unpredictability is a required property of an RBG. It should not be possible to predict the output of a properly implemented and working RBG. Forward secrecy refers to the inability to predict future output of the RBG based on the knowledge of previous output values and/or internal states. The inability to determine prior output of an RBG, given knowledge of the current or any future output of the RBG, is known as backward secrecy.

The decision whether to incorporate backward and/or forward secrecy is determined by the requirements of the consuming application.

The following factors should be considered when deciding to incorporate backward and/or forward secrecy.

1. In some instances, achieving backward secrecy is more important than achieving forward secrecy. For example, if a cryptosystem is stolen, an adversary may attempt to read the old messages processed by that system. Forward secrecy is not really a concern, since the system is no longer in use by the original owner. Achieving backward secrecy is straightforward (for example by the appropriate use of a one-way function in the design), although there may be a performance cost associated with providing this property, depending on the design.

2. Trying to achieve forward secrecy may not be appropriate for some cryptosystems. For example, a smart card may be initialised at the point of manufacture with sufficient entropy in the seed and is set to expire after a limited time (e.g., two or three years). In this case, it may be much easier to replace the card with a new smart card that is seeded with a different seed than it is to build forward secrecy into the RBG design.

3. In some instances, achieving forward secrecy may be more important than achieving backward secrecy. Consider, for example, the secure generation of nonces. It is not necessary for a random bit generation algorithm to have backward secrecy as all of the previous outputs will be known. However, forward secrecy may be useful to prevent an adversary with knowledge of the generator from being able to predict later outputs.

5.2 Requirements of an RBG

The following requirements apply to all RBGs, both deterministic and non-deterministic.

The requirements provided below are fundamental to the security of cryptographic mechanisms that require random input.

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

1. Under reasonable assumptions, it shall not be feasible to distinguish the output of the RBG from true random bits that are uniformly distributed. Informally, all possible outputs occur with equal probability and a series of outputs appears to conform to the uniform distribution.

2. Given a sequence of output bits, it shall not be known to be feasible to compute or predict any other output bit, either past or future.

3. Throughout the lifetime of the RBG, it shall not be possible to predict output stream sequence repeats.

4. The RBG shall not leak relevant secret information (e.g., internal state of a DRBG) through the output of the RBG.

5. The RBG shall not leak secret information from the perspective of an adversary.

NOTE 1 An example of when this could happen would be a timing attack.

6. 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 International Standard and the requirements of the consuming application.

7. On detection of an error, the RBG shall either (a) enter a permanent error state, or (b) 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 innately in the design.

NOTE 2 See 8.8 and 9.8 for NRBG and DRBG information on RBG errors and health tests.

8. 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 I).

9. The probability that the RBG can “misbehave” in some pathological way that violates the output requirements (e.g., constant output or small cycles, i.e., looping such that the same output is repeated) shall be sufficiently small. That means that the probability of error should be consistent with the overall confidence in correct operation that is required of the RBG, which need not be the same as the required strength of cryptographic security.

10. 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).

11. The implementation shall be designed to allow validation, including specific design assertions about what the RBG is intended not to do. The validation of an NRBG means that the NRBG behaves as expected, not just during normal operation, but also at the boundaries of the intended operational conditions. Security-relevant branches in the code that govern behaviour in exceptional conditions (e.g. initialisation, failed health tests, etc.) shall be validated by deliberately forcing all error conditions to occur during validation testing.

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

5.3 Optional requirements for an RBG

Optional requirements for an RBG are as follows.

1. If the RBG is capable of operating in more than one mode, the RBG should return information about the mode in which it is operating, upon request.

2. A consuming application may require that the RBG be run in a test mode, for example, when using a seed that is non-secret and therefore is capable of reconstructing the bits it generates. When an RBG is in the test mode, the RBG shall not be capable of being used to generate secret bits. When using a secret seed, the RBG shall not operate in the test mode.

3. Considered as a glass box, an RBG should have backward secrecy. If this attribute is supported it 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 predict any previous output bit.

4. Considered as a glass box, an RBG may have forward secrecy. If supported, it means that given all accessible information about the RBG (comprising some subset of inputs, algorithms, and outputs), it shall be infeasible (up to the specified security strength) to compute or predict any future output bit at the time that 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 should achieve without constraining or mandating how it is 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 in this International Standard.

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 herein, that RBG might still conform to this International Standard if the RBG component is used within a system that supplies the missing elements and satisfies the remaining requirements. Such an RBG will be said to possess residual functional requirements. Those residual requirements become a system prerequisite for use of the RBG.

NOTE There exist RBGs, which do not incorporate all functional components as depicted in Figure 1. Physical NRBGs, for instance, 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 may be the case where all functional requirements are met without incorporating all basic components. They are:

a) the entropy source;

b) additional inputs;

c) the internal state;

d) the internal state transition function;

e) the output generation function; and

f) support functions.

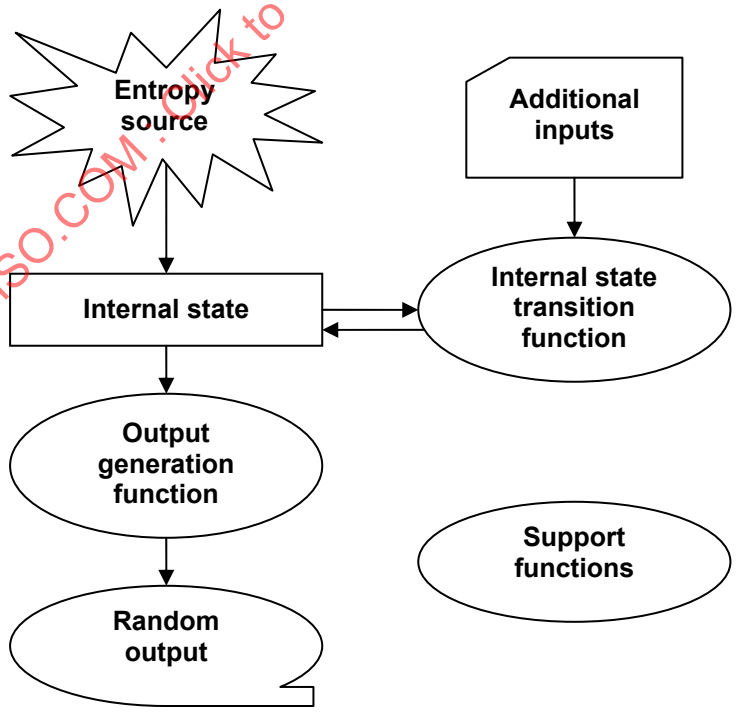

Figure 1 — RBG functional model

6.2.2 Entropy source

6.2.2.1 Entropy source overview

The entropy source is the source of unpredictable bits. These bits may be biased and may be dependent on one another to some extent; in fact this will often be the case. In the case of a DRBG, the entropy source may be a distinct, and possibly a remote NRBG. However, for an NRBG, the entropy source component is the combination of the existence of some entropy producing activity, the detection of this activity and the digitisation mechanism.

The entropy source shall produce bits with non-zero entropy. It is the only model component that produces entropy. 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 entropy 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, other than that the entropy source has non-zero entropy. However, any particular RBG design will necessarily establish specific conditions that shall hold for the design to work.

The model allows the entropy source to be decomposed in different ways. One way is recursive decomposition. In particular, it is possible that the entropy source for one RBG is another, distinct (i.e., external) RBG. Indeed it is possible that the external RBG uses a further RBG as an entropy source. There is no innate limit on such a recursive construction, however it should be clear that such a recursion shall ultimately end with a true (i.e., non-deterministic) entropy source without which the whole construction is invalidated. Furthermore, the construction should make sure that there is sufficient true entropy within the system to support the input criteria for each of the individual RBGs.

6.2.2.2 Requirements for an entropy source

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

1. The entropy source shall be based upon well-established principles, or extensively characterised 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 will reliably obtain or exceed 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 unauthorised external body shall be minimized according to the requirements of the application.

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

6.2.3 Additional inputs

6.2.3.1 Additional inputs overview

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

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. It may include a seed, a counter, a clock value, user input, 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. Since the internal state transition functions and the output generating function are deterministic functions, and since deterministic functions have the property that they always give the same output for the same input, it may 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.

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 portion of the internal state that supports this functionality is known as the working state. Care should be taken to ensure that the value of the working state cannot be influenced by any outside event (for example, by restarting the RBG).

The second role of the internal state is to parameterise the deterministic internal state transition functions and/or output generation function in such a way that when this parameter is unknown, the deterministic function exhibits the properties required of it. For example, it may parameterise 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 (sometimes known as the random bit generator's key) 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 not required that all RBGs have a secret parameter.

NOTE 2 Key management concerns are discussed in ISO/IEC 11770-1[6].

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 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-initialised).

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 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 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 is available to an adversary, size $2x$ should be used (see e.g. [10]).

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. See ISO/IEC 11770-1 for further details on 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 An optional requirement for the internal state

The internal states that are used to produce public data, (e.g., nonces and initialisation 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 entropy source nor any other RBG data inputs are considered to act directly on the internal state. Instead, the entropy source and other RBG data inputs are arguments to functions that act on the internal state.

Since the internal state may 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 may 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 the entropy 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 (for example, 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 entropy source input.

In an NRBG, the internal state transition functions determine how the entropy collection affects the internal state of the RBG and may work in a variety of ways: these functions may be simple, stateless algorithms, or they may 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 function shall be verifiable via a known-answer test.

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

3. The internal state transition function 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 function to return to a prior state in normal operation (this excludes testing and authorised verification of the RBG output).

6.2.5.3 An optional requirement for the internal state transition functions

The internal state transition function may enable the RBG to recover from the compromise of the internal state (i.e., provide forward secrecy) through 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 may 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 may be as simple as the identity function or as complex as a cryptographic function. The OGF may format or block the output to conform to an external interface convention. Often, the OGF is a "one-way" function, so that it will be hard to invert the OGF in order to recover part of the internal state.

The OGF is deterministic, and will always produce 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 is 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 required 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.

6.2.6.3 An optional requirement for the output generation function

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

6.2.7 Support functions

6.2.7.1 Support function overview

The support functions are concerned with assessing the health of the RBG: they do not change the internal state. Support functions include functions that assess the entropy of the input, functions that assess the statistical quality of the output, and functions that 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 by known-answer tests. The "health" of the entropy source and the quality of the output needs to be checked using statistical techniques or tests.

Detailed information on assessing the health of NRBGs and DRBGs can be found in clauses 8 and 9 respectively.

6.2.7.2 Requirements for the support function

The requirements for support functions of an RBG are as follows.

1. An RBG shall be designed to permit testing that will ensure that 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 operations while in the error state.

7 Types of RBGs

7.1 Introduction to the types of RBGs

Every RBG shall have a primary entropy source. RBGs are classified into two basic types depending on the nature of their primary entropy source. The primary entropy source will either be a non-deterministic entropy source or a deterministic entropy source.

If an entropy source is 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 a non-deterministic entropy source. In particular, no deterministic algorithm should be able to predict the output of a non-deterministic entropy source.

If an entropy source is a seed value, it is called a deterministic entropy source. This seed value (see 9.3) may be supplied to the RBG from an external system or internally from the RBG. If the seed value is supplied externally to the RBG, it is assumed by the RBG to have sufficient entropy and to have been drawn from a specified set. On the other hand, if the seed value is supplied internally from within the RBG, it is assessed to have sufficient entropy and to have been drawn from a specified set.

Care shall be taken to ensure that an adversary cannot gain sufficient control over the seed value to degrade the entropy of the output of the RBG.

An RBG is said to be an NRBG if its primary entropy source is non-deterministic. An RBG is said to be a DRBG if its primary entropy source is deterministic. An RBG may also utilise additional entropy sources that can be either deterministic or non-deterministic.

7.2 Non-deterministic random bit generators

NRBGs can be further specified into sub-classes depending on the nature of their non-deterministic entropy source. A non-deterministic entropy source will be either physical or non-physical.

A physical non-deterministic 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 power is applied to the measuring device. Examples of physical non-deterministic entropy sources 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, non-deterministic entropy source is any non-deterministic entropy source that is not a physical non-deterministic entropy source. Examples of non-physical, non-deterministic entropy sources include measuring the time between key presses or sampling of data in regularly used portions of 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 non-deterministic entropy source. A non-physical NRBG is an RBG with a non-physical non-deterministic entropy source.

A (physical or non-physical) NRBG is said to be “pure” if all of its entropy sources are non-deterministic.

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) which are typical for DRBGs.

The advantage of using a seed value as an additional entropy source to the NRBG is that it allows the output of the NRBG to be parameterised by the user. Hence, the output of the NRBG can be individualised and may even allow different users to use the same non-deterministic entropy source without compromising the security of the system. Even if there are not several users who use the same entropy source, a secret seed value may be viewed as an additional security feature.

NOTE 1 Additional information on classes of RBGs is found at [5]

NOTE 2 A secret seed value, for instance, could be an additional DRBG-typical security feature. If the NRBG meets the requirements of the standard even with a simple state transition function and a simple output function (cf. 8.2 and E.1) a cryptographic state transition function or a cryptographic output function could be an additional DRBG-typical security feature.

7.3 Deterministic random bit generators

DRBGs can also be classified as either pure or hybrid. A DRBG is said to be “pure” if all of its entropy sources are seeds and “hybrid” if it uses a non-deterministic entropy source as an additional entropy source.

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).

NOTE 1 The security of a hybrid NRBG is based on its non-deterministic part while the security of hybrid DRBGs (see 9.3.4) is essentially based on their deterministic components. If an RBG may be viewed as both a hybrid NRBG or alternatively as a hybrid DRBG we denote it is as “hybrid RBG”. Loosely speaking, hybrid RBGs have two security anchors.

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

NOTE 2 Additional information on classes of DRBGs is found at [4]

7.4 The RBG spectrum

The NRBG and DRBG classes are depicted in Figure 2.

The distinction between an NRBG and a DRBG can be seen as a classification 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 or the inclusion of non-deterministic entropy sources in 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 correct choice of RBG is a cost/benefit trade-off depending on the application requirements.

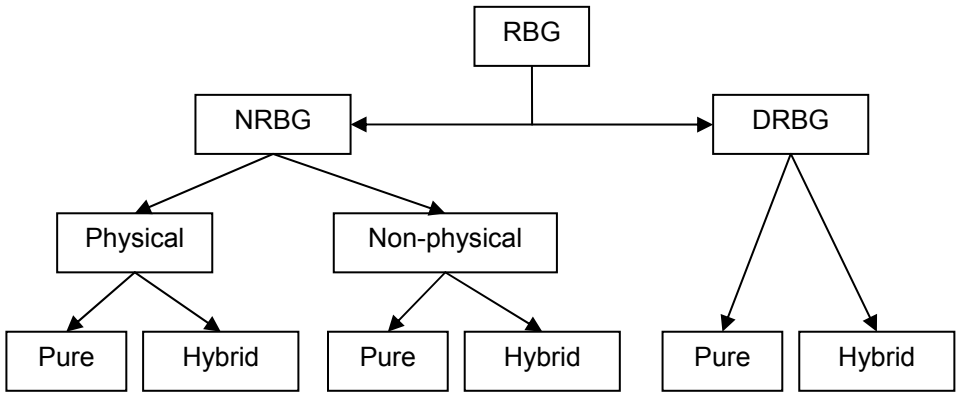

Figure 2 — RBG classes

8 Overview and requirements for an NRBG

8.1 NRBG overview

It is likely that in the majority of applications, the choice will be to use a DRBG for random bit generation. Thus, it is envisioned that the primary use of an NRBG satisfying this International Standard will be to generate random initial seeds for a DRBG. However, this International Standard does not preclude the use of an NRBG to generate all of the random bits needed by the application.

For the purposes of this International Standard, an NRBG shall meet the requirements specified in clauses 5 and 6. However, NRBG unique objectives and requirements, beyond those specified in clauses 5 and 6, which will be imposed on NRBG designs, are detailed in sections 8.3 through 8.9. NRBG examples can be found at Annex E.

8.2 Functional model of an NRBG

This clause will introduce a specialisation of the components and the general model, for the case of an NRBG. It will describe the general operation of an NRBG, including the objectives each NRBG functional component is intended to accomplish.

Figure 3 illustrates a functional block diagram depicting a conceptual NRBG that satisfies this International Standard. In this diagram, dashed lines indicate a component that is optional, depending on various factors. It is important to note that the components shown do not necessarily have to be implemented as actual physical components, but do need to be implemented functionally in some sense.

Each of the components, and the objectives and requirements on these components, 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 will be required in an NRBG. In some applications, there may be a legitimate argument that none of the requirements for a given component are applicable. If this can be adequately justified and documented, that component may 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 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 this entropy source data with the working state data to produce a new working state. The internal state transition function accomplishes this distillation by associating with each of the possible

combinations of input sequence and current working state a value of the next working state, as determined by the current value of the secret parameter. If the length of the entropy source input data used for each internal state transition is n bits 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, since 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. The distinguishing characteristic between a DRBG and an NRBG is that an NRBG shall add new entropy to its state at a rate greater than or equal to the rate that it outputs entropy. Therefore, whenever the consuming application requires random output, the NRBG ensures that the internal state contains sufficient entropy that has not yet been used to produce random output, and then uses the output generation function to process the current working state to produce random output.

The output generation function takes the internal state, which may need to be kept confidential from an adversary, and produces an output that cannot be distinguished from random. The output generation function will typically also be 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 will produce uniformly distributed binary output.

For many physical entropy sources the entropy per bit is sufficiently large so that these bits are either directly applicable or applicable after a simple post processing operation (e.g., XORing the entropy source output with the current internal state). Moreover, the behaviour of physical entropy sources can often be described using a stochastical model (see e.g. [7]). Such a model allows accurate statistical testing designed to detect the failure of the source (i.e., when the source produces output which is not sufficiently random for the application). It may therefore be recommendable to choose very simple state transition functions and output generation functions, which facilitates to determine the stochastical behaviour of the output bits. Secret parameters may not be necessary in such a case.

For non-physical entropy sources, the entropy per bit may be lower than for physical entropy sources and it may be difficult to accurately determine the moment when the entropy source fails. In such cases, more complex post processing operations may 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 will 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 health tests on the various components. Undetectable failures or compromises are addressed in two ways. First, the internal NRBG state includes a periodically changing secret parameter that parameterises the deterministic operation of the internal state transition function. Because of this component, knowledge of the remaining part of the internal state (known as the working state) of the NRBG and all the inputs to the NRBG is insufficient to determine the NRBG output. The second approach is the inclusion of a safety margin in the maintenance of entropy during NRBG operation, so that decreases in 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 meeting this International Standard 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.

The requirement that the NRBG should continue to operate in a manner no less secure than a DRBG in the event that the entropy source completely fails may be dropped if:

1. it can be verified that the health tests on the entropy source(s) will sufficiently soon afterwards detect possible weaknesses and failures of the entropy source(s) which would deteriorate the quality of the random bits in an unacceptable manner; and

2. appropriate measures are initiated in such a case (e.g. stopping the generation of random bits).

NOTE It is usually feasible to meet these requirements for physical entropy sources as they use dedicated hardware. A generic proposal is discussed in [13].

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

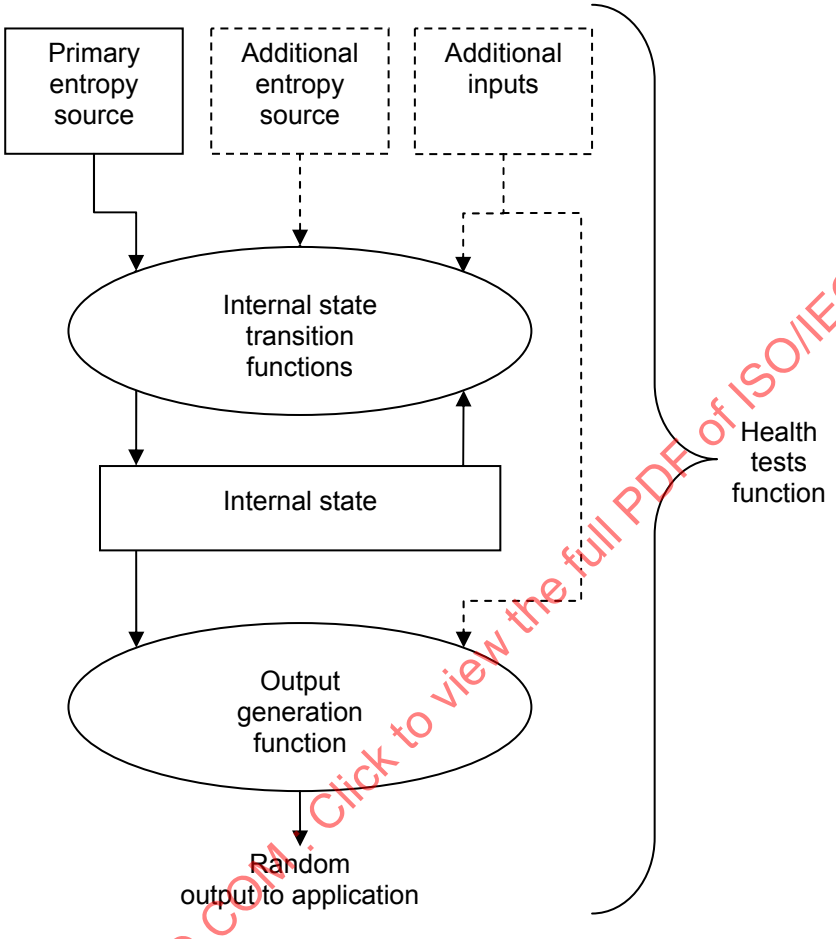

Figure 3 — Block diagram of an NRBG

8.3 NRBG entropy sources

8.3.1 Primary entropy source for an NRBG

8.3.1.1 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 solely on the unknown initial seed for unpredictability. In an NRBG, unpredictability is based on the use of one or more sources of entropy. These sources are known as non-deterministic entropy sources and can be further classified as either physical or non-physical.

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. Examples of physical entropy sources include measuring the time between radioactive emissions of an unstable substance, and the output of a noisy diode, which receives a constant input voltage level and outputs a continuous, normally distributed analogue voltage level.

A non-physical entropy source is any non-deterministic entropy source that is not a physical entropy source. Examples of non-physical entropy sources are system data or RAM content 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 can be found in 8.3.2 and 8.3.3 respectively.

Depending on the entropy source, the output of the entropy source may not be adequate for direct use as random 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 NRBG by processing the entropy source output with an internal state transition function that produces binary output data.

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

8.3.1.2 Requirements for an NRBG primary entropy source

The primary entropy source is the foundation of the non-deterministic behaviour of the NRBG. By the very definition of an NRBG, the 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., it shall not be definable by any known algorithmic rule.

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

3. The entropy 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 on the entropy source.

4. Failure or severe degradation of the entropy source shall be detectable. Based on the frequency of health tests being performed, this detection may not be 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 produces falls below a threshold value.

8.3.1.3 Optional requirements for an NRBG primary entropy source

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

1. The entropy source shall be stationary in a mathematical sense. If it is not stationary, then there shall 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 will greatly complicate the process of estimating the rate of entropy contribution and increase the difficulty of validating the resulting NRBG design (unless the design includes additional entropy sources that do satisfy this requirement).

2. Appropriate health tests tailored to the known statistical model of the source should place special emphasis on detection of behaviour 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.

8.3.2 Physical entropy sources for an NRBG

8.3.2.1 NRBG physical entropy source overview

A physical entropy source will typically provide a continuous stream of output whilst it has power applied. However, this output is not necessarily provided as a simple binary string. For example, if the physical entropy source is based on the time between radioactive emissions the entropy source may not provide a simple binary output string that contains entropy, but as a series of timings between emissions. Alternatively, if the physical entropy source used is based on voltage variations in a noisy diode then the output could be a series of voltage measurements.

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

Since a physical entropy source is likely to be contained within a cryptographic boundary of an NRBG, the emphasis of the security conditions are on the proper collection and interpretation of a true physical entropy source and not on the possibility that an adversary may be able to gain information and/or initiate undue influence of the source.

An advantage of a physical entropy source is that it is generally feasible to model as a stochastic process that can be handled. This in turn, leads to efficient online health checks that accurately determine the moment when a physical entropy source fails. Such tests can be used to prevent a physical NRBG from continuing to operate with a predictable entropy source, and may remove some of the need for complex state transition or output generation functions.

8.3.2.2 Requirements for an NRBG physical entropy source

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

1. The threshold values shall be chosen so that the output string contains a sufficient amount of entropy for the application. Note that there is a difference between the entropy displayed by a source (which may produce data at infinite precision) and the entropy provided by a binary interpretation of that source.

2. The total failure of the entropy source shall be immediately detectable. A degradation of the entropy source shall be detected sufficiently fast, where “sufficiently fast” depends on the degree of degradation.

8.3.2.3 An optional requirement for an NRBG physical entropy source

The entropy source should be formally analysed and the threshold values chosen in such a way that the output contains the greatest possible amount of entropy. This is possible because true physical entropy sources are comparatively simple compared to human entropy sources.

8.3.3 NRBG non-physical entropy sources

Non-physical entropy sources are usually provided by a system upon request of the NRBG. Hence, non-physical entropy sources are typically outside the defined protection boundary of the NRBG (see 5.4). The data is usually already binary in nature. For example, the source could 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 entropy source accurately as a stochastic process, it may be more difficult to determine whether such a source is operating correctly or not. Hence, non-physical NRBGs often use complex state transition and output

generation functions that guarantee that the RBG will be at least as secure as a DRBG for the period of time between the entropy source failing and the failure being detected.

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

8.3.4 NRBG additional entropy sources

8.3.4.1 NRBG additional entropy sources overview

The operation of an NRBG may also include one or more additional entropy sources. An additional entropy source may 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 characterised statistical model.

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

Finally, having multiple entropy sources may 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 have an additional entropy source that is not a non-deterministic entropy source but rather a deterministic entropy source, i.e., a seed value. An NRBG that also has a deterministic entropy source is known as a hybrid NRBG. Hybrid NRBGs are discussed in 8.3.5.

Despite providing additional unpredictability to the output of an RBG, the security of the RBG shall rest solely upon the primary entropy source. Hence, the output of an RBG shall remain secure even if the output of all of the additional entropy sources are known to an adversary and/or if an adversary has a certain measure of influence over the output of the additional entropy sources.

8.3.4.2 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:

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

b) 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 satisfy the same requirements as the primary entropy source. Alternatively, instead of including an additional entropy source, it may be acceptable to solve this problem by using the NRBG only for initial seed generation for a DRBG; and

c) 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 as an entropy source. In this case, the additional entropy source shall satisfy the conditions given in 8.3.5.

8.3.4.3 Optional requirements for NRBG additional entropy sources

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

1. 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 entropy source), or a completely different component or process, this source should 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 facilitates the design and evaluation or independent verification processes by allowing the primary and secondary entropy

sources to be analysed separately, and also reduces the likelihood of a failure in the primary entropy source.

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

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

8.3.5 Hybrid NRBGs

An NRBG is considered a hybrid NRBG if it e.g. takes a deterministic source as an additional input. The main advantage of a hybrid NRBG is that its unpredictable output is now parameterised by a seed value. This may allow for extra security as knowledge of the output of the primary entropy source may now not be enough to allow an adversary to break the scheme, or it may allow a single non-deterministic entropy source to be used by several RBGs with different seed values without compromising security.

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

1. An adversary should not be able to predict the next bit with the probability significantly greater than one half, even if the attacker has complete control over the seed.

2. No adversary should be able to recover any information about the seed value by observing the output of the RBG.

3. No unauthorised person should be able to manipulate, influence or update the seed value.

8.4 NRBG additional inputs

8.4.1 NRBG additional inputs overview

The operation of an NRBG may require it to take certain public and/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 a correctly authenticated and authorised person or 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 requirement of any and 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 shall provide uninterruptible power which is a system issue and beyond the scope of this International Standard.

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 mandatory requirement; however it is not compulsory that any portion of the internal state depend upon previous states, i.e., there is no mandatory 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 compromise of the random output by an adversary who has knowledge of or influence on the entropy source.

The internal state will consist 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 may 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, which 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 could 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 authenticated and authorised person or personnel within the boundary of the operational policy. More information on the issues associated with control of secret parameter is provided in ISO/IEC 11770-1[6].

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 A 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 has been designed to detect unauthorised penetration attempts.

3. The initial value of the secret parameter, if it exists, shall contain sufficient entropy to meet the security requirement of the application. This initial secret parameter can 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 to be combined in some way with the entropy source data (i.e., this additional entropy serves as a temporary secondary entropy source).

NOTE 2 Examples of user interaction might 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 (for example, “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 forward and backward secrecy. The secret parameter should only be updated through commands issued by properly authenticated and authorised personnel within the boundary of the operational security policy.

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 to 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) fail and the working state becomes compromised, the NRBG shall resist any attempts to force it to produce predictable output.

NOTE 4 This could be achieved by either ensuring that the NRBG ceases operation when the entropy source(s) fail, or, if the NRBG includes a secret parameter, by ensuring that the NRBG continues to operate in a manner no less secure than a DRBG. In particular, the secret parameter should be of sufficient length to resist any form of cryptanalytic attack against the NRBG, including exhaustive search.

8.5.3 Optional requirements 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 sources fail or become 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 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 having sufficient entropy at each power-up initialisation without having to immediately create a new secret parameter value. If this is done, it shall be preserved in a way that protects it from adversarial access.

NOTE This protection could 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.

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 will typically work in two parts.

1. It will carry the secret parameter over without change.

2. It will update 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.

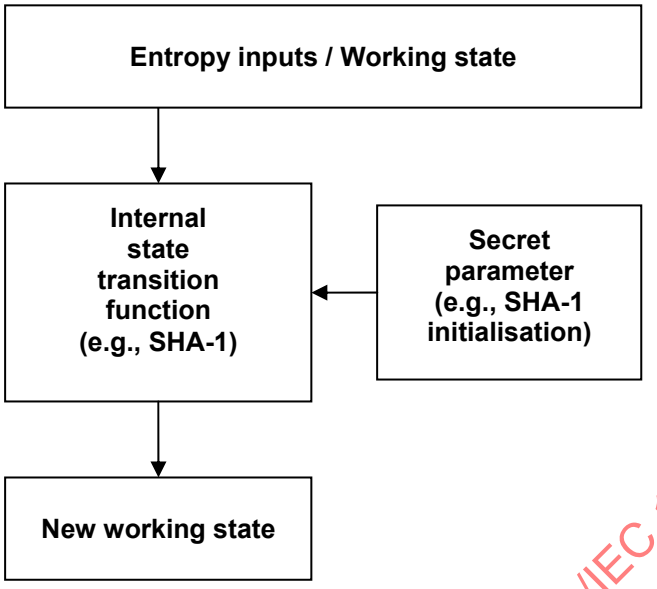

Figure 4 — Example of the internal state transition function

NOTE In Figure 4, the internal state transition function is SHA-1 and the secret parameter is the 160 bit application specific constant for SHA-1.

8.6.2 Requirements for the NRBG internal state transition functions

The use of an internal state transition function is mandatory only in that it needs to collect and store the output of the entropy sources. If this is the only function that the internal state transition functions provide then there are no additional requirements that the internal state transition functions shall satisfy.

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 a 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 rated security strength (in bits) of the NRBG.

8.6.3 Optional requirements 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 may 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 of the bits in the working state and the entropy source input shall influence the internal state transition functions output bits.

3. The operation of the internal state transition function should be protected against observation and analysis via power consumption, timing, radiation emission, or other side channel attacks. The values that the internal state transition function operates on (internal state, secret parameter and entropy source input) are the critical values upon which confidentiality of random output is based. Side channel analysis could 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 working state and, possibly, secret parameter), and 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.

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 may 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.

8.7.3 An optional requirement for the NRBG output generation function

The following requirement is optional. It may be the case that in certain designs the inclusion of this requirement is highly recommended.

The output generation function should not be able to be inverted to reveal 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 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 initialisation, without intervention by other applications, processes, or users, and may also be requested by the user at any time.

The health tests presented in the following text include three sets of tests; tests on the deterministic components of the NRBG, tests on 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 may be cases where the rate of input entropy or random output is too low to feasibly implement all of the 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.[1)]

General requirements and requirements that apply to each of these sets of tests are presented in the following four sub-clauses. In some cases, the NRBG may have additional features or functionality not addressed by this International Standard, and may include specialised 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 need to be performed depend 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 guaranteed by frequent testing of the entropy source, the deterministic component may be simplified.

8.8.2 General NRBG health test requirements

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

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

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 may be used as random output following the completion of all health tests.

NOTE An exception to this requirement may exist if health tests are continually applied to the NRBG (e.g., health tests on the entropy source or statistical tests on the 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 initialisation shall include an integrity check on the implementation code (RAM, 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 should require the performing of the health tests with increased frequency during an operational session without serious degradation in performance, in addition to during power-up initialisation (a reasonable interval is one based on the frequency of requests for random bits).

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 initialise 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 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) For the purposes of this International Standard, 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 may be found at [8].

1. The known-answer tests shall be included in the overall health tests performed at each power-up and/or re-initialisation, 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 length for known-answer test values.

3. The known-answer tests shall include the entire NRBG, including not only each component of the NRBG, but also the overall NRBG control components. This may be done by setting the internal state to a fixed pattern, blocking the entropy source data and replacing the entropy source with a fixed bit sequence, and running the NRBG process in its operational mode. This action will produce an output sequence of at least the length determined according to point #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 that has successfully passed the known-answer health tests shall not be used as random output following the completion of all health tests. The current value of the NRBG internal state following successful completion of the known-answer test shall not be used.

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 on entropy sources

The objective of these tests is to detect variation in the behaviour of the entropy source from the intended behaviour. Since the entropy source is assumed to not produce unbiased, independent binary data in the vast majority of cases, traditional randomiser tests (e.g., monobit, chi square, and runs tests that test the hypothesis of unbiased, independent bits) will virtually always fail and thus not be useful. In general, tests on the entropy sources will have to be tailored carefully to the entropy source, taking into account the non-uniform statistical behaviour of the correctly operating entropy source.

For non-deterministic entropy sources obeying very complicated statistical models, and for non-physical entropy sources in particular, it may not be feasible to develop statistical tests that correspond precisely to a statistical model of the entropy source. In these cases it may 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. Selection of the patterns used for such tests should take into account the entropy source's likely failure behaviour.

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

1. The tests on the entropy sources shall be included in the overall health tests performed at each power-up and/or re-initialisation, at periodic intervals during use, and on demand.

NOTE 1 For example, the periodic interval for tests on the entropy source could be determined by the module's implementation policy.

2. As a minimum, each entropy 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. (Constant outputs are those consisting only of a single value of the digitised entropy 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 will depend on the characteristics of the entropy source, and shall be chosen such that when the entropy 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 entropy 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 entropy sources.

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

NOTE 2 To recover or exit from an error state, an NRBG will be required to follow its maintenance procedures.

Although optional, it is recommended that the tests on each entropy source should include tests that take into account the known characteristics of the entropy source.

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. In general, the inclusion of the internal state transition function and output generation function results in the health tests on the random output from an NRBG playing a smaller role than they would if the tests were applied directly to output from a non-deterministic entropy source. These functions will 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 requirement that the NRBG continue to operate as 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 address mainly statistical tests. As indicated above, usually, statistical tests are ineffective to validate 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 and/or re-initialisation, 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 out generation function. See 9.8.8 for continuous RBG test specifics.

3. If any of the health tests on the random output 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-initialisation), 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 will be required to follow its maintenance procedures.

Optionally is the following health test on the random output. When this recommendation is chosen, the health test shall meet functional requirement #1 above. 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 is considered 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 3 These four tests are rooted in FIPS 140-2 [9] and also adapted in AIS-31[5].

a. Monobit test: Let X be the number of ones in the sample. The test is passed if $9725 < X < 10275$.

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 $0 \le i \le 15$). Evaluate the following: $X = \frac{16}{5000} \sum_{i=0}^{15} f(i)^2 - 5000$. The test is passed if $2.16 \le X \le 46.17$.

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 of either zeroes or ones. The test is passed if there are no long runs.

NOTE 4 The rationale for the runs and long runs tests is discussed in Annex J.

Table 1 — Runs test intervals

Run Length	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 may be performed on the random output if:

1. it is deemed that greater assurance is required. If such is the case then the tests in the optional requirement 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

2. 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 three. If the random output passes the test during this set of attempts, the NRBG can resume normal operation.

8.9 NRBG component interaction

8.9.1 NRBG component interaction overview

The components of an NRBG described in this International Standard are designed to accomplish certain security objectives by satisfying the specific objectives and requirements. This section presents some additional requirements involving the 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, with the additional constraint that the output generation function shall not reuse data from the working state. Note that 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 could be much larger than its minimum acceptable length (i.e., 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 entropy into the working state after each application of the internal state transition function, regardless of the entropy contribution from the others. This allows independent safety margins on the maintenance of entropy within the NRBG.

8.9.3 Optional requirements 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 operate and interact as a DRBG. A natural way to accomplish this is by designing the NRBG as a DRBG 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 influence from the entropy sources even when the output generation function does not require new working state data (this could 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, during which the internal state might be more vulnerable to observation due to the increased length of time that 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 pseudorandom sequence of bits from an initial value called a seed, along with possible other inputs. Because of the deterministic nature of the process, a DRBG is said 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, the seed and any other input information. However, if the input is kept secret, and the algorithm is well designed, the bit strings will appear to be random.

A DRBG is classified into two types depending on its inputs. Although a DRBG shall take a seed value as its primary entropy source, it may also take additional entropy sources as inputs. If these additional entropy sources are all deterministic (i.e. seed values) then the DRBG is called a pure DRBG. If at least one of these

entropy sources is non-deterministic, then the DRBG is called a hybrid. A hybrid DRBG shall satisfy all the requirements of a pure DRBG plus it shall also satisfy some extra security requirements discussed in 9.3.4.

At any given time after a seed has initialised a DRBG, a DRBG exists in a state that is defined by all prior input information. The primary 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 entropy sources).

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

The seeds for a DRBG require that the entropy be provided by an entropy input source (e.g., an NRBG). The security of an implementation that uses a DRBG is a system implementation issue; both the DRBG and its entropy input source shall be considered.

For the purposes of this International Standard, a DRBG shall meet the requirements specified in Clause 5 and 6. DRBG unique objectives and requirements, beyond those specified in Clause 5 and 6, which will be imposed on DRBG designs are detailed in 9.3 – 9.9. DRBG examples can be found in Annex C.

9.2 Functional model of a DRBG

This section will introduce a specialisation of the components and general model, for the case of a DRBG. It will describe the general operation of a DRBG, including the objectives each DRBG functional component is intended to accomplish.

Figure 5 provides a functional block diagram for a DRBG that satisfies this International Standard. It is important to note that the components shown do not necessarily have to be implemented as actual separate program routines, but do need to be implemented functionally in some sense. Each of the components, and the objectives and requirements on these components, 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 will be required in a DRBG. In some applications, there may be a legitimate argument that none of the requirements for a given component are applicable. If this can be adequately justified and documented, that component may be omitted from the DRBG, 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. For example, there are two common scenarios in which a DRBG may 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 (time, number of calls, etc.) and the internal state is suitably protected (say, by protective hardware).

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

The seed can be directly supplied in one of two ways. The first way to directly supply a seed would be if the seed were produced by an NRBG that meets the requirements of this International Standard (i.e., an approved NRBG). The second way would be if the seed 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 is indirectly supplied if the seed 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 specialised internal state transition function referred to as seeding. Access to this specialised internal state transition function should only be available to authorised parties, and 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 will, on demand, do two things: it will update its internal state and output a block of random appearing data. The internal state will be updated by a deterministic internal state transition function, referred to as *updating internal state* in Figure 5, that takes as input the current internal state, any additional inputs supplied by the user and the outputs of any non-deterministic entropy source. In particular, it will not take the original seed as input unless the seed is held in the internal state specifically for this purpose. The output of the DRBG will be computed by a deterministic output generating function, which takes the current (just updated) internal state as input.

It is often useful to think of the internal state as being the combination of a working state that is continually updated, and 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 will indeed often be cryptographic keys). The control of these secret parameters is outside the scope of this International Standard.

NOTE 1 Key management guidance may be found at ISO/IEC 11770. [6]

Typically, a DRBG will also contain a mechanism to update the internal state in the event that a new seed value is supplied. Such an update will be performed by a specialised internal state transition function and is known as reseeding. The internal state after reseeding may or may not depend upon the previous internal state depending on the reseeding method but may merely involve clearing the DRBG's working state and loading the new seed in a similar manner to the initial seeding operation.

A secure DRBG will also include 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.

Each of the DRBGs specified 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 provides backward secrecy.

In the case of a DRBG, forward secrecy depends on the use of a secret seed and the insertion of additional input during each generation process with sufficient entropy to meet the security requirements. Forward secrecy may be provided during one instance of a DRBG by the addition of user input. However, the degree of forward secrecy depends on the amount of entropy introduced by the user input. If the user input introduces entropy that is at least equal to twice the strength of the DRBG for every request for pseudorandom bits, then "full" forward secrecy is usually provided.

NOTE 2 For example, if a DRBG is to provide 128 bits of security, then to have the full forward secrecy, we would need both the seed and each additional input to have at least 128 bits of entropy.

This may not be practical for many applications. However, user input with even a small amount of entropy provides some degree of forward secrecy. It may be appropriate for an application to require user 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 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 could 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.

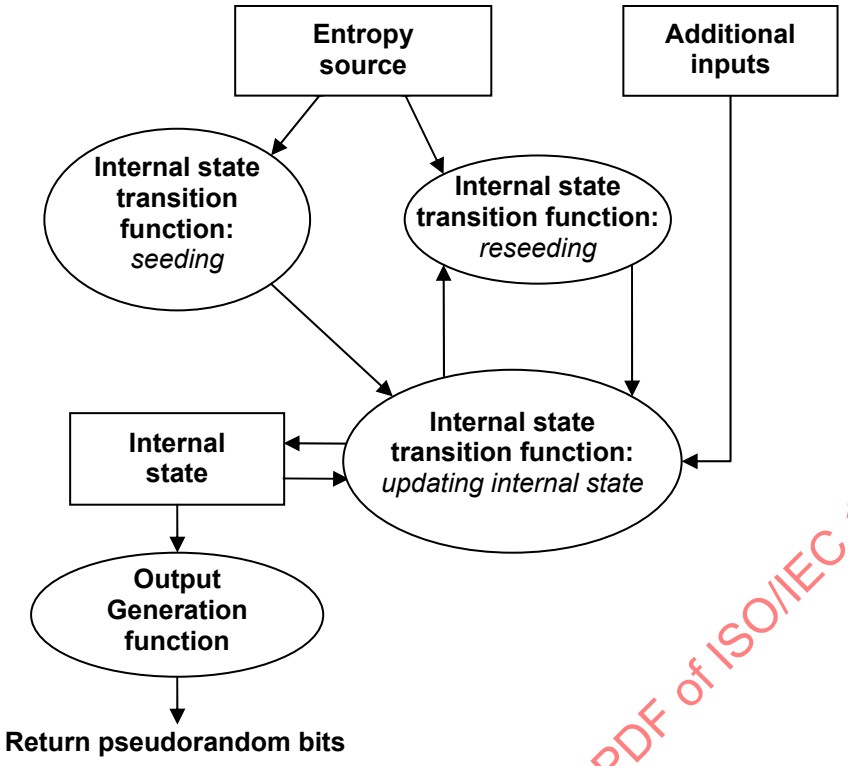

Figure 5 — DRBG model

9.3 DRBG entropy source

9.3.1 Primary entropy source for a DRBG

The primary entropy source for a DRBG is a seed value. This seed value shall be drawn from an entropy source with a given entropy output rate such as an NRBG, 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, see 9.3.2.

The seed, seed 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 International Standard does not require full entropy for a seed but does require sufficient entropy, the length of the seed may be greater than the security strength specified, in order to accommodate the needed entropy. That is, the seed length shall, at minimum, be the number of bits in the specified security strength, however, the seed length should be more than the minimum in order to increase assurance of sufficient entropy and address any concerns of possible reuse of a seed. Note that using the minimum seed length is acceptable only if the seed is generated by a full entropy source, such as an NRBG or DRBG seeded by an NRBG. As the general case is that the seed may be of variable size, the specification of the DRBG algorithms make this assumption, although a specific implementation may be optimised to support a specific length seed.

Entropy analysis for a seed is a critical component to the security assurance associated with the DRBG. The importance of entropy analysis is most readily demonstrated when something goes wrong. For example, if the security strength of 80 bits is desired and believed to be achieved, but only 40 bits is actually achieved, this amount of entropy is exhausted quite easily. Even if the actual entropy is 60 bits, a determined adversary could still exhaust the entropy at a reasonable cost.

The generation or entry of entropy into a DRBG using an insecure method may 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 entropy input and should include a personalisation string (see 9.4 for further information on personalisation string) when the source of entropy is not considered strong enough. The combination of the entropy input and the optional personalisation string is called the seed material. A derivation function shall be used to distribute the entropy input across the entire seed (e.g., so that the seed is not constructed with all the entropy on one end of the seed) whenever a personalisation string is used, or a personalisation string is not used and the entropy input is not independent and uniformly distributed throughout the entropy input string. Whether or not the personalisation string is present, the resulting seed shall be unique.

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 1 If the DRBG has backward and forward secrecy and if the implementation is resistant against side-channel attacks one may neglect 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 initialise 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 entropy: The entropy input for the seed shall contain sufficient entropy for the desired level of security, and the entropy shall be distributed across the seed. A consuming application may or may not be concerned about collision resistance. In order to accommodate possible collision concerns, a seed shall have entropy that is equal to or greater than 120 bits or the required security strength for the consuming application; whichever is greater (i.e., entropy ≥ max (120, *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 size: The minimum size of the seed depends on the selected DRBG, the security strength required by the consuming application and the entropy source. The seed size shall be at least equal to the required security strength in bits and may be larger, depending on the entropy rate and assurances thereof of the entropy source (see above discussion). For example, if 160 bits of entropy are required, the quality of the entropy source may necessitate a seed size of 240 bits or more to achieve the 160 bits of entropy.

5. Seed privacy: Seeds 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), unauthorised entities may be able to 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 may not be possible, and reseeding may actually reduce security. In these cases, the best policy might 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 should happen to become 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 NRBG (or chain of DRBGs that is initiated by an NRBG) used to provide the new seed. Generating too many outputs from a seed (and other input information) may provide sufficient information for successfully predicting future outputs. Periodic reseeding will reduce 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 shall be checked to assure 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 (i.e., non-repeating) seeds should be used for the generation of different types of random data (i.e., the "instances" of the DRBGs should be different). For example, the seed used to generate public values should be different than the seed used to generate secret values. The seed used by a DRBG technique to generate asymmetric key pairs should be different than a seed used by the same (or a different) DRBG technique to seed other DRBGs, which should, in turn, be different than a seed used by the same (or a different) DRBG technique to generate symmetric keys. The seed used by a DRBG technique to generate random challenges should be different than the seed used by the same (or a different) DRBG technique to generate PINS or passwords. However, the amount of seed separation is a cost/benefit decision.

NOTE 2 If the DRBG has backward and forward secrecy and if the implementation is resistant against side-channel attacks one may use the random number generated by one instance of the DRBG for different types of random data without loss of security.

9.3.2 Generating seed values for a DRBG

A seed value for an approved DRBG shall be produced in one of three ways.

1. The seed shall be produced by an approved NRBG that produces output at a sufficient entropy rate.

2. The seed shall be produced by an approved DRBG that produces output at a sufficient entropy rate. This DRBG shall also have a seed, which shall be produced in accordance with these seed generation requirements. Hence, a chain of DRBGs can be envisaged that produce seeds for the next DRBG. However, this chain shall always begin with either an approved NRBG or an approved DRBG that produces its own seed values. In other words, if a seed is produced by an approved DRBG or chain of approved DRBGs, the highest level DRBG has to get its seed from an NRBG or a DRBG with an internal NRBG.

3. The seed shall be generated by an appropriate entropy source. This source may be biased and/or produce dependent bits. If a seed value 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 entropy sources for a DRBG

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

A DRBG may have an additional entropy source that is not a deterministic entropy source but rather a non-deterministic entropy source. A DRBG that also has a non-deterministic entropy source is known as a hybrid DRBG. Hybrid DRBGs are discussed in 9.3.4.

Despite providing additional unpredictability to the output of an RBG, the security of the RBG shall rest solely upon the primary entropy source. Hence, the output of an RBG shall remain secure even if the output of all of the additional entropy sources are known to an adversary and/or if an adversary has a certain measure of influence over the output of the additional entropy sources.

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

9.3.4 Hybrid DRBG

A DRBG is considered a hybrid DRBG if it takes a non-deterministic entropy source as an additional input; otherwise it is considered a pure DRBG.

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

1. An adversary should not be able to predict the next bit with the probability significantly greater than one half, even if the attacker has complete control over output of the non-deterministic entropy source.

2. No adversary should be able to recover any information about the non-deterministic entropy source by observing the output of the RBG.

3. No unauthorised person should be able to manipulate or influence the non-deterministic entropy source.

9.4 Additional inputs for a DRBG

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

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

DRBGs in this International Standard allow for the use of an optional personalisation string during instantiation. A personalisation string is used in conjunction with entropy bits to produce a seed. Examples of data that may be included in a personalisation 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 initialised with a new seed, the counter may 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 should 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 for the DRBG instance, 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 personalisation 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, i.e., even if the attacker can adaptively choose the values of the additional inputs, it should be unable to predict the next bit produced by the RBG with probability significantly greater than one half.

5. It shall be ensured that the internal state transition function that updates the internal state has the property that the size of its range does not degenerate after repeated calls to it without intermediate reseeding, as this would exhaust the entropy of the internal state.

9.5 Internal state for a DRBG

The internal state is the memory of the DRBG and 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, user 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 during each iteration of the DRBG.

The internal state can be thought of as a combination of a working state, which is potentially updated during every execution, and a secret parameter, which is constant during every execution and only updated periodically by the outside intervention of the user.

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. 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. In the event that there are values that the secret parameter should not take (for example, “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.

9.6 Internal state transition function for a DRBG

The internal state transition function uses the internal state and one or more algorithms to produce pseudorandom bits. During this process, the internal state of the DRBG is altered. The algorithms used and the method of altering the internal state depends on the specific DRBG.

The DRBGs in this International Standard have three separate internal state transition functions, which are that:

1. prior to the initial use of the DRBG, seed material is obtained, and all initial input is determined. The initial input is used to determine the initial state of the DRBG;
2. each request for pseudorandom bits produces the requested bits using the current internal state and determines a new internal state that is used for the next request; and
3. when an application determines that reseeding of the DRBG is required, a reseeding function obtains new seed material, combines it with the current internal state values, and determines a new internal state for the next request for pseudorandom bits. By combining the new seed material with the current internal state, the entropy available from the current state is not lost, but is enhanced by the entropy of the new seed material.

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

a. 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, reseeding and/or rekeying may need to be performed more frequently.

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

c. The internal state transition functions should have the property that all of the bits in the working state (and the non-deterministic entropy source input, if it exists) influence the internal state transition functions output bits.

d. The operation of the internal state transition function should be protected against observation and analysis via power consumption, timing, radiation emission, or other side channel attacks. The values that the internal state transition function operates on (internal state, secret parameter, and entropy source input) are the critical values upon which confidentiality of future random output is based. Side channel analysis could potentially defeat this confidentiality.

e. It shall be ensured that the internal state transition function that updates the internal state has the property that the size of its range does not degenerate after repeated calls to it without intermediate reseeding, as this would exhaust the entropy of the internal state.

9.7 Output generation function for a DRBG

The output generation function of a DRBG produces pseudorandom bits that are a function of 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 input information. 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 is available.

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 may 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.

9.8 Support functions for a DRBG

9.8.1 DRBG support functions overview

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

A DRBG shall be designed so as to permit testing that will ensure 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 operations while in an error state and all output shall be inhibited.

NOTE Error states may include “hard” errors that indicate an equipment malfunction that may require maintenance, service, repair or replacement of the DRBG, or may include recoverable “soft” errors that may require initialisation or resetting of the DRBG. Recovery from error states should be 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 may also be conducted

continually. An RBG may optionally perform other health tests for DRBG functionality in addition to the tests specified in this International Standard.

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 are successful.

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 may 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.

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., an authentication code, digital signature algorithm, or HMAC) shall be applied to all software and firmware (e.g., within 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.

9.8.7 DRBG manual key entry test

When security information is manually entered into a DRBG (e.g., a seed 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 EDC or that the duplicate entries match. 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 DRBG continuous random bit generator test

A DRBG shall be tested for failure to a constant value. If each call produces blocks of n bits (where $n \geq 80$), the first block generated after power-up, initialisation or reset shall not be used, but shall be saved for

comparison with the next block to be generated. Each subsequent generation of an n bit block shall be compared with the previously generated block. The test shall fail if any two compared n bit blocks are equal. If the test fails, the DRBG shall enter an error state and output an error indicator.

If each call to the generator produces fewer than 80 bits, then the first n bits (for some $n \geq 80$) generated after power-up, initialisation or reset shall not be used, but shall be saved for comparison with the next n generated bits. Each subsequent generation of n bits shall be compared with the previously generated n bits. The test fails if two compared n bit sequences are equal. If the test fails, the DRBG shall enter an error state and output an error indicator.

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 a source outside the DRBG boundary), or the DRBG may be designed to generate keys from seed material. The use of externally provided keys may 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 specified 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 be at least the required security strength of the consuming application. For example, when 112 bits of security are required by an application, the key(s) shall have at least 112 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 required strength of the consuming application.

    b) For DRBGs that use multiple seeds to determine a DRBG instance, each seed 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 seeds). The combination of all seeds shall have entropy that is equal to or greater than the required strength of the consuming application.

5. Keys provided from an external source: Keys generated externally shall have full entropy (i.e., each bit of a key shall be independent of every other bit of the key) and shall be generated using an NRBG or a DRBG (or chain of DRBGs) that is seeded by an NRBG.

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 NRBG (or chain of DRBGs that is initiated by an NRBG) used to provide the new key. In some implementations (e.g., smartcards), an adequate rekeying process may not be possible, and rekeying may actually reduce security. In these cases, the best policy might be to replace the DRBG, obtaining a new key in the process (e.g., obtain a new smart card).

a) Generating too many outputs using a given key may provide sufficient information for successfully predicting future outputs. Periodic rekeying will reduce 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), unauthorised entities may be able to 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 should use different keys.

Annex A
(normative)

Combining RBGs

Combining RBGs is specifically allowed by this International Standard, as long as they are combined in an acceptable way. Combination 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 can be combined with another RBG if it can be shown that the RBGs are not correlated with each other. For example, this might be because they use a different seed and/or use a different algorithm. 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 can be combined with an unapproved RBG not specified in this International Standard 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 exclusive OR (XOR) then this condition is satisfied.

Combining an approved RBG with an unspecified RBG might be done to allow gaining the advantages of the unspecified RBG that were thought to be superior to approved alternatives in some quality which were not specified in this International Standard. It also could be done when one forum requires use of one method and another forum requires use of another method.

EXAMPLE 1 A generator with a physical entropy source may use a properly initialised 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 should not be possible to influence the flow of bits from the NRBG to the DRBG except, perhaps, for operational testing or product validation.

Annex B
(normative)

Conversion methods

B.1 Random number generation

B.1.1 Techniques for generating random numbers

This International Standard 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 $0 \le a_i \le r\text{-}1$, 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 $(0 \le s \le r\text{-}1)$; and

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

In this section, four techniques by which such sequences of random numbers can be generated from sequences of random bits are specified.

If the range of the number required is not $0 \le a_i \le r\text{-}1$ 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 $0 \le a_i \le b\text{-}a$.

B.1.2 The simple discard method

Let m be the unique positive integer satisfying $2^{m-1} \le r \le 2^m - 1$ (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 will always satisfy $0.5 \le r/2^m < 1$. 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 below is recommended.

B.1.3 The complex discard method

Choose a small positive integer t, and then let m be the unique positive integer satisfying $2^{m-1} \le r^t \le 2^m - 1$ (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 $0 \le a_i \le r\text{-}1$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 will always satisfy $0.5 \le r^t/2^m < 1$. Hence, given r, it is recommended to choose t so that t is the smallest value such that $r^t/2^m$ is close to 1. For example, if r = 3, then choosing t = 3 means that m = 5 and $r^t/2^m$ = 27/32 ≈ 0.84, and choosing t = 5 means that m = 8 and $r^t/2^m$ = 243/256 ≈ 0.95.

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

B.1.4 The simple modular method

Let m be the unique positive integer satisfying $2^{m-1} \le r \le 2^m - 1$, 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 $(0 \le s \le r\text{-}1)$ 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 $1/r$ will be negligible. A value of l= 64 is recommended.

B.1.5 The complex modular method

Choose a small positive integer t, and then let m be the unique positive integer satisfying $2^{m-1} \le r^t \le 2^m\text{-}1$ (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 $c = \sum_{i=0}^{m+l-1} 2^i b_i \bmod r^t$

Let $(a_0, a_1, \ldots, a_{t-1})$ be the unique sequence of values satisfying $0 \le a_i \le r\text{-}1$ 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 will be negligible for a large enough security parameter. A value of l = 64 is recommended.

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

B.2 Extracting bits in the Dual_EC_DRBG

B.2.1 Potential bias in an elliptic curve over a prime field F_p

Occasionally, a random number less than a specific modulus is produced as an output. It is necessary to be able to extract unbiased bits from this number modulo the modulus. This method examines the modulus M and the pseudorandom number r as bits from left to right, with the highest numbered bit on the left, then partitions the possible values of r into disjoint sets based on the largest number of random bits that might be extracted.

As a tiny illustrative example, if M = 11, then the binary representation of M is 1011, and the possible values of r (in binary) are: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, and 1010, for a total of 11 possible values. The method proceeds as follows:

Let $M(x)$ represent the x-th bit of M and let $r(x)$ represent the x-th bit of r.

1. Since the 4th bit of M is 1, look at the 4th bit of r.

   a) The 4th bit of r is 0 in eight cases: 0000, 0001, 0010, 0011, 0100, 0101, 0110, and 0111. Each possible 3-bit pattern is present for the next 3 bits (bits 3-1). Therefore, as $M(4)$ = 1 and $r(4)$ = 0, the rightmost 3 bits of r can be extracted as unbiased bits, and the extraction process is complete.

   b) The 4th bit of r is 1 in three cases: 1000, 1001 and 1010. Since all patterns are not represented in the next three bits of the subset of r remaining (i.e., only 000, 001 and 010 are present), there is a bias. Go to step 2, considering only these three cases.

2. The 3rd bit of M is 0, and the 3rd bit of r is always 0 for the three cases. Therefore, there is a bias, go to step 3, considering only the rightmost 2 bits of M and r for the three cases.

3. The 2nd bit of M is 1; look at the 2nd bit of r for the three cases (i.e., 1000, 1001 and 1010).

   a) The 2nd bit of r is a 0 in two cases: 1000 and 1001. Each possible case is present in the next bit (i.e., bit 1). Therefore, as $M(2)$ = 1 and $r(2)$ = 0, a single bit (i.e., bit 1) can be extracted as an unbiased bit, and the process is complete.

   b) The 2nd bit of r is 1 in one case: 1010. Since all possible 1 bit patterns are not possible in bit 1, there is a bias, and no bits should be extracted.

For this tiny example, 8/11 of the time three unbiased bits can be extracted, 2/11 of the time one unbiased bit can be extracted, and 1/11 of the time no unbiased bits can be extracted. As can be seen, it is not possible to know in advance exactly how many random bits will be able to be extracted for a particular r, but the average over many values for r is able to be determined.

The general case is as follows. Let both M and the random r values have m bits, i.e., $M(m)$ = 1, although $r(m)$ may be either 0 or 1.

1. Do $i = m$ to 1 by –1

{

If ($M(i)$ = 1) and ($r(i)$ = 0), then go to step 3.

NOTE There is a skew if $M(i)$ = 0 or $r(i)$ = 1. (i-1) bits cannot be extracted.

}

2. Failure. Not even one unbiased bit can be extracted. Quit or try again for a new value of r.

3. Extracted bits = Rightmost (i-1) bits of r.

4. Success

The method just described does an excellent job of extracting unbiased bits from a number x in the range [0, p). For all of the recommended mod p curves (specified in D.1) this algorithm amounts to extracting the rightmost m-d bits, where d is the position of the leftmost 0 in x, and p has m bits. Here d =1 is referencing the leftmost bit; thus at least 1 bit is always deleted from the output.

Each recommended prime has at least 32 leading 1's in its binary representation, leaving a tiny fraction of x for which extracting the rightmost m-d bits based solely on the position of the first 1 will result in bits with a negligible bias: < $2^{-(31+d)}$ in the d-th bit.

B.2.2 Adjusting for the missing bit(s) of entropy in the x coordinates

While the algorithm described in B.2.1 removes biases in the extracted bits, it does not address the issue of entropy: it should not be possible to predict any bits from previously observed bits. Unfortunately with the **Dual_EC_DRBG (…)** there will always be some "missing" entropy in a block of output, but it can easily be made small enough to become negligible. In particular, the amount of information remaining will in no way undermine the strength of the pseudorandom bits.

To illustrate, suppose a mod p curve with m = 256 is selected, and that all 256 bits produced were output by the generator, i.e., that blocksize = 256 bits. In addition, suppose that 255 of these bits are published, and the 256th bit is kept "secret." About 50% of the time, the unpublished bit could easily be determined from the other 255 bits. Similarly, if 254 of the bits are published, about 25% of the time the other two bits could be predicted. This is a simple consequence of the fact that only about 1/2 of all 2^m bit strings of length m occur in the list of all x coordinates of curve points.

The situation is slightly worse with the binary curves, since each has a cofactor of 2 or 4. This means that only about 1/4 or 1/8, respectively; of the m bit strings occur as x coordinates. Thus the recommended elliptic curves (in Annex D) have m bit outputs, which are lacking 1, 2 or 3 bits of entropy, when taken in their entirety.

The "abouts" in the preceding example can be made more precise, taking into account the difference between 2^m and p, and the actual number of points on the curve (which is always within $2p^{(1/2)}$ of p). For the recommended curves these differences won't matter at the scale of the results, so they will be ignored. This allows the heuristics given here to work for any curve with "about" (2^m)/f points, where f = 1, 2 or 4 is the curve's cofactor.

The basic assumption needed is that the approximately (2^m)/(2f) x coordinates that do occur are "uniformly distributed", i.e., a randomly selected m bit pattern has a probability 1/(2f) of being an x coordinate. The assumption allows a straightforward, albeit approximate, calculation for the entropy in the rightmost (least significant) m-d bits of **Dual_EC_DRBG** output, with d << m.

The formula is: $E = -\sum_{j=0}^{2^d} \left[2^{m-d}\, binomprob(2^d, z, 2^d - j) \right] p_j \log_2 p_j$.

The above term represents the approximate number of (m-d) bit strings which fall into one of 1+2^d categories as determined by the number of times j it occurs in an x coordinate; z = (2f-1)/2f is the probability that any particular string occurs in the x coordinate; $p_j = \frac{\mathrm{f}(j)}{2^{(m-1)}}$ is the probability that a member of the j-th category occurs. Note that the j = 0 category contributes nothing to the entropy (randomness).

B.2.3 Values for E

The values of E for d up to 16 are: (cof = $\log_2 f$).

$\log_2 f$: 0 d: 0 entropy: 255.00000000 m-d: 256

$\log_2 f$: 0 d: 1 entropy: 254.50000000 m-d: 255

$\log_2 f$: 0 d: 2 entropy: 253.78063906 m-d: 254

$\log_2 f$: 0 d: 3 entropy: 252.90244224 m-d: 253

$\log_2 f$: 0 d: 4 entropy: 251.95336161 m-d: 252

$\log_2 f$: 0 d: 5 entropy: 250.97708960 m-d: 251

$\log_2 f$: 0 d: 6 entropy: 249.98863897 m-d: 250

$\log_2 f$: 0 d: 7 entropy: 248.99434222 m-d: 249

$\log_2 f$: 0 d: 8 entropy: 247.99717670 m-d: 248

$\log_2 f$: 0 d: 9 entropy: 246.99858974 m-d: 247

$\log_2 f$: 0 d: 10 entropy: 245.99929521 m-d: 246

$\log_2 f$: 0 d: 11 entropy: 244.99964769 m-d: 245

$\log_2 f$: 0 d: 12 entropy: 243.99982387 m-d: 244

$\log_2 f$: 0 d: 13 entropy: 242.99991194 m-d: 243

$\log_2 f$: 0 d: 14 entropy: 241.99995597 m-d: 242

$\log_2 f$: 0 d: 15 entropy: 240.99997800 m-d: 241

$\log_2 f$: 0 d: 16 entropy: 239.99998900 m-d: 240

$\log_2 f$: 1 d: 0 entropy: 254.00000000 m-d: 256

$\log_2 f$: 1 d: 1 entropy: 253.75000000 m-d: 255

$\log_2 f$: 1 d: 2 entropy: 253.32398965 m-d: 254

$\log_2 f$: 1 d: 3 entropy: 252.68128674 m-d: 253

$\log_2 f$: 1 d: 4 entropy: 251.85475372 m-d: 252

$\log_2 f$: 1 d: 5 entropy: 250.93037696 m-d: 251

$\log_2 f$: 1 d: 6 entropy: 249.96572188 m-d: 250

$\log_2 f$: 1 d: 7 entropy: 248.98298045 m-d: 249

$\log_2 f$: 1 d: 8 entropy: 247.99151884 m-d: 248

$\log_2 f$: 1 d: 9 entropy: 246.99576643 m-d: 247

$\log_2 f$: 1 d : 10 entropy: 245.99788495 m-d: 246

$\log_2 f$: 1 d: 11 entropy: 244.99894291 m-d: 245

$\log_2 f$: 1 d: 12 entropy: 243.99947156 m-d: 244

$\log_2 f$: 1 d: 13 entropy: 242.99973581 m-d: 243

$\log_2 f$: 1 d: 14 entropy: 241.99986791 m-d: 242

$\log_2 f$: 1 d: 15 entropy: 240.99993397 m-d: 241

$\log_2 f$: 1 d: 16 entropy: 239.99996700 m-d: 240

$\log_2 f$: 2 d: 0 entropy: 253.00000000 m-d : 256

$\log_2 f$: 2 d: 1 entropy: 252.87500000 m-d: 255

$\log_2 f$: 2 d: 2 entropy: 252.64397615 m-d: 254

$\log_2 f$: 2 d: 3 entropy: 252.24578858 m-d: 253

$\log_2 f$: 2 d: 4 entropy: 251.63432894 m-d: 252

$\log_2 f$: 2 d: 5 entropy: 250.83126431 m-d: 251

$\log_2 f$: 2 d: 6 entropy: 249.91896704 m-d: 250

$\log_2 f$: 2 d: 7 entropy: 248.96005989 m-d: 24

$\log_2 f$: 2 d: 8 entropy: 247.98015668 m-d: 248

$\log_2 f$: 2 d: 9 entropy: 246.99010852 m-d: 247

$\log_2 f$: 2 d: 10 entropy: 245.99506164 m-d: 246

$\log_2 f$: 2 d: 11 entropy: 244.99753265 m-d: 245

$\log_2 f$: 2 d : 12 entropy: 243.99876678 m-d: 244

$\log_2 f$: 2 d: 13 entropy: 242.99938350 m-d: 243

$\log_2 f$: 2 d: 14 entropy: 241.99969178 m-d: 242

$\log_2 f$: 2 d: 15 entropy: 240.99984590 m-d: 241

$\log_2 f$: 2 d: 16 entropy: 239.99992298 m-d: 240

B.2.4 Observations

The following observations should be noted.

1. Each table starts where it should, at 1, 2 or 3 missing bits.
2. The missing entropy rapidly decreases.
3. Each doubling of the $\log_2 f$ factor requires about 1 more bit to be discarded for the same level of entropy.
4. For $\log_2 f = 0$, i.e., the mod p curves, d=13 leaves 1 bit of information in every 10000 (m-13) bit outputs.

Based on these calculations, for the mod *p* curves it is recommended that an implementation remove the leftmost, i.e., most significant 13 bits of every m bit output, and that the **Dual_EC_DRBG (…)** be reseeded every 10000 iterations. For the binary curves, either 14 or 15 bits should be removed, as determined by the cofactor being 2 or 4, respectively.

Using this value for d in the mod p curves insures that no bit has a bias exceeding $1/2^{44}$.

For ease of implementation the value of d should be adjusted upward, if necessary, until the number of bits remaining, m-d = *blocksize*, is a multiple of 8. By this rule the actual number of bits discarded from each block will range from 16 to 19.

Annex C
(normative)

DRBGs

C.1 DRBG mechanism examples

This annex contains DRBG mechanism examples that meet the requirements of this International Standard. This is not a complete list. Other mechanisms are possible if they meet the requirements stated in the main body of this International Standard.

The DRBGs within this annex are given in pseudocode.

NOTE Some pseudocode examples contained herein also appear at [3].

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 has been designed to use any ISO/IEC cryptographic hash-function, and may be used by applications requiring various levels of security, provided that the appropriate hash-function is used and sufficient entropy is obtained for the seed. These hash-functions shall be as specified in ISO/IEC 10118-3.

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 an ISO/IEC hash-function 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 F) depending on the hash-function used.

The length of the seed (*seedlen*) shall be at least the maximum of the hash output block size (*outlen*) and the security strength.

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 shall be used throughout. The hash-function to be used shall meet or exceed the desired security strength of the consuming application.

Table C.1 provides an example of an ISO/IEC approved cryptographic hash-function, illustrating security strength, required minimum entropy and seed length.

Table C.1 — Security strength table for hash functions

Hash-function	**SHA-1**	**SHA-224**	**SHA-256**	**SHA-384**	**SHA-512**
Supported Security Strengths	80, 112, 128	80, 112, 128, 192	80, 112, 128, 192, 256	80, 112, 128, 192, 256	80, 112, 128, 192, 256
Required Minimum Entropy	**max**(120, Security Strength)				
Seed Length (*seedlen*)	440	440	440	888	888
NOTE The description of the function **max**(…) is given in C.2.2.2.1.					

C.2.2.2 Description

C.2.2.2.1 General

The instantiation and reseeding of **Hash_DRBG (…)** consists of obtaining an entropy input with at least the requested amount of entropy by the consuming application. The entropy input 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 *entropy_input* 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 forward secrecy capability is required by the DRBG; and
7. (Optional) A transformation of the entropy input using a one-way function for later comparison with new entropy input when the DRBG is reseeded; this value shall be 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.
C	A *seedlen*-bit constant that is calculated during the instantiation and reseeding processes.
data	The *data* to be hashed.
entropy_input	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 an entropy 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 an ISO/IEC hash-function.
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.
min_entropy	The minimum amount of *entropy* to be obtained from the entropy source and provided in the *seed*.
min_length	The minimum length of the *entropy_input*.
Null	The null (i.e., empty) string.
outlen	The length of the hash-function output block
personalisation_string	A *personalisation string*.
prediction_resistance_flag	A flag indicating whether or not forward secrecy requests should be handled. *prediction_resistance_flag* = 1 = allow = Allow_prediction_resistance, 0 = do not allow = No_prediction_resistance.
prediction_resistance_request_flag	Indicates whether or not forward secrecy is required during a request for pseudorandom bits. *Prediction_resistance_request_flag* = 1 = Provide_prediction_resistance, 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.
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.
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 will be used to create the seed.
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*, *seedlen*, *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.
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 shall be used to instantiate the **Hash_DRBG (…)** process.

Let **Hash (…)** be the ISO/IEC hash-function to be used, and let *outlen* be the output length of that hash-function.

Instantiate_Hash_DRBG (…):

Input : integer (*requested_strength*, *prediction_resistance_flag*), string *personalisation_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 five security strengths.

   If (*requested_strength* ≤ 80), then *strength* = 80

   Else 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**(120, *strength*).

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

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

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

7. *seed_material* = *entropy_input* || *personalisation_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*).

NOTE 2 Precede V with a byte of zeroes.

11. *reseed_counter* = 1.

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

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

Get_entropy (…):

The specific details of the **Get_entropy (…)** process are left to the implementer. A high level example is provided below.

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

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

Process:

1. Obtain *entropy_input* with *entropy* ≥ *min_entropy* and with the appropriate *length* from the appropriate source, where *min_length* ≤ *length* ≤ *max_length*.

2. **Return** ("Success", *entropy_input*).

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: bitstring *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 3 *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 4 If an implementation does not handle all five security strengths, then step 2 of **Instantiate_Hash_DRBG(…)** will be modified accordingly.

NOTE 5 If no *personalisation_string* will ever be provided, then the *personalisation_string* parameter in the input may be omitted, and step 7 of **Instantiate_Hash_DRBG(…)** becomes *seed_material* = *entropy_input*.

NOTE 6 If an implementation does not need 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(…)**) may be omitted.

C.2.2.2.3 Reseeding Hash_DRBG (…) Instantiation

The following process or its equivalent shall be used to reseed the **Hash_DRBG (…)** process. Let **Hash (…)** be the ISO/IEC hash-function to be used, 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 a *state* 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** (120, *strength*).

4. *min_length* = *min_entropy*.

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

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

7. *seed_material* = 0x01 || V || *entropy_input* || *additional_input*.

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

NOTE This step combines the new *entropy_input* with a fixed byte, the entropy present in V and with any *additional_input* provided; then distribute 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 shall be used to generate pseudorandom bits. Let **Hash (…)** be the ISO/IEC hash-function to be used (ISO/IEC 10118-3), 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*, bitstring *pseudorandom_bits*, where *status* = "Success" or a failure message.

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, 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*, bitstring V.

Output: bitstring *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) mod $2^{seedlen}$.

5. *pseudorandom_bits* = Leftmost *requested_no_of_bits* of W.

6. **Return** (*pseudorandom_bits*).

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

NOTE 2 If an implementation does not need the *prediction_resistance_flag*, then the reference to the *prediction_resistance_flag* in Hash_DRBG may 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** (120, *strength*) new bits of entropy, which are added to the state.

2. utilising the automatic reseeding feature of the DRBG. If the maximum number of updates for the state is reached, the DRBG will call 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 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 Frequent calls to the **Get_entropy (…)** function may 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, which is based on any ISO/IEC cryptographic hash-function found in ISO/IEC 10118-3. 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 shall be used throughout an **HMAC_DRBG** instantiation. The hash-function used shall meet or exceed the security requirements of the consuming application.

C.2.3.2 Description

C.2.3.2.1 General

The instantiation and reseeding of **HMAC_DRBG (…)** consists of obtaining a *seed* with the appropriate amount of entropy. The entropy input 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 cryptographic hash-funciton);

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 a prediction resistance capability is required for the DRBG.

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

additional_input	Optional additional input
data	The *data* to be hashed.
entropy_input	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 an entropy 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 that shall be as specified in 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.
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.
min_entropy	The minimum amount of *entropy* to be obtained from the entropy source and provided in the *seed*.
min_length	The minimum length of the *entropy_input*.
Null	The null (i.e., empty) string.
outlen	The length of the cryptographic hash-function output block
personalisation_string	A personalisation string.
prediction_resistance_flag	A flag indicating whether or not forward secrecy requests should be handled. *prediction_resistance_flag* = 1 = allow = Allow_prediction_resistance, 0 = do not allow = No_prediction_resistance.
prediction_resistance_request_flag	Indicates whether or not forward secrecy is required during a request for pseudorandom bits. *prediction_resistance_request_flag* = 1 = Provide_prediction_resistance, 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.
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.
seed_material	The data that will be used to create the seed.
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*. Note that for this DRBG mechanism, the **HMAC_DRBG_Update** function also serves as a derivation function for the instantiate 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 Table C.1 in this annex.

HMAC_DRBG_Update (…):

Input: bitstring (*provided_data*, *K*, *V*).

Output: bitstring (*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 shall be 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 that keyed hash-function.

Instantiate_HMAC_DRBG (…):

Input: integer (*requested_strength*, *prediction_resistance_flag*), string *personalisation_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 five security strengths.

   If (*requested_strength* ≤ 80), then *strength* = 80

   Else 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**(120, *strength*).

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

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

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

7. *seed_material* = *entropy_input* || *personalisation_string*.

8. *Key* = 0x00 00…00.

9. V = 0x01 01…01.

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 shall be 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 a *state* 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** (120, *strength*).
4. *min_length* = *min_entropy*.
5. (*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_length*, *max_length*).
6. If (*status* ≠ "Success"), then **Return** (Failure message).
7. *seed_material* = *entropy_input* || *additional_input*.
8. (*Key*, V) = **HMAC_DRBG_Update**(*seed_material*, *Key*, V).
9. Update the appropriate *state* values.
   - 9.1. *state*(*state_handle*).V = V.
   - 9.2. *state*(*state_handle*).*Key* = *Key*.
   - 9.3. *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 shall be 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*, bitstring *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*, *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* may be omitted as an input parameter, and step 8 may 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 will never provide *additional_input*, then a *Null* string replaces the *additional_input* in step 9.2.

   9.3 If (*status* $\neq$ "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* $\neq$ *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*.

    16.1 *state*(*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 International Standard have been designed to use any ISO/IEC approved block cipher algorithm and may be used by applications requiring various levels of security. These block cipher algorithms shall be as specified in ISO/IEC 18033-3. 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 an ISO/IEC approved block cipher, illustrating the security strengths, entropy and seed requirements that shall be 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	80, 112, 128	128	256
AES-192	80, 112, 128, 192	192	320
AES-256	80, 112, 128, 192, 256	256	384

C.3.2 CTR_DRBG

C.3.2.1 Discussion

The **CTR_DRBG (…)** uses an approved block cipher algorithm in the counter mode and shall be as specified in ISO/IEC 10116. The same block cipher algorithm and key length shall be used for all block cipher operations. The block cipher algorithm and key size shall meet or exceed the security requirements of the consuming application.

C.3.2.2 Description

C.3.2.2.1 General

The instantiation and reseeding of **CTR_DRBG (…)** consists of obtaining a *seed* with the appropriate amount of entropy. The entropy input 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 shall be ≤ *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.
blocklen	The length of the block cipher algorithm's output block.
entropy_input	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 an entropy 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 shall be ≤ 2^{35} bits (for a 128 bit block cipher, there are 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 shall be ≤ 2^{35} bits for 128 bit block ciphers, and ≤ 2^{19} bits for 64 bit block ciphers.

min_entropy	The minimum amount of entropy to be obtained from the entropy source and provided in the *seed*.
Null	The null (i.e., empty) string.
personalisation_string	A personalisation string.
prediction_resistance_flag	A flag indicating whether or not forward secrecy requests should be handled; *prediction_resistance_flag* = 1 = allow = Allow_prediction_resistance, 0 = do not allow = No_prediction_resistance.
prediction_resistance_request_flag	Indicates whether or not forward secrecy is required during a request for pseudorandom bits; *prediction_resistance_request_flag* = 1 = Provide_prediction_resistance, 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 shall be $\leq 2^{32}$ for 128 bit block ciphers, and $\leq 2^{16}$ for 64 bit block ciphers.
seedlen	The length of the seed, where *seedlen* = *blocklen* + *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 shall be used to initially instantiate the **CTR_DRBG (…)** process.

Instantiate_CTR_DRBG (…):

Input: integer (*requested_strength*, *prediction_resistance_flag*), string *personalisation_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* ≤ 80), then (*strength* = 80; *keylen* = 128)

   Else 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 five security strengths, and determines the key length.

3. *seedlen* = *blocklen* + *keylen*.

4. *temp* = **len** (*personalisation_string*).

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

NOTE 2 If a *personalisation_string* will never be provided, then the *personalisation_string* input parameter and steps 4 and 5 can be omitted.

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*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_entropy*, *max_length*).

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

   6.4 *seed_material* = *entropy_input* || *personalisation_string*.

   NOTE 3 If no *personalisation_string* is provided in step 6.4, then step 6.4 may be omitted, and step 6.5 becomes *seed_material* = **Block_Cipher_df** (*entropy_input*, *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 to be 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 *personalisation_string* = *personalisation_string* || $0^{seedlen - temp}$.

NOTE 4 If the *personalisation_string* is too short, it will be padded with zeroes.

7.4 *seed_material* = *entropy_input* ⊕ *personalisation_string*.

NOTE 5 If no *personalisation_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 = $0^{blocklen}$.
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 need 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) may 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 shall be *seedlen* bits in length. The following or an equivalent process shall be used as the **Update (…)** function.

Update (…):

Input: integer *keylen*, bitstring (*seed_material*, *Key*, V).

Output: bitstring (*Key*, V).

Process:

1. *seedlen* = *blocklen* + *keylen*.
2. *temp* = *Null*.
3. While (**len** (*temp*) < *seedlen*) do:

   3.1 V = (V + 1) mod $2^{blocklen}$.

   3.2 *output_block* = **Block_Cipher** (*Key*, V).

   3.3 *temp* = *temp* || *output_block*.

4. *temp* = Leftmost *seedlen* bits of *temp*.
5. *temp* = *temp* ⊕ *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. bitstring *input_string*: The string to be operated on.

2. integer *no_of_bits*: the number of bits returned by **Block_Cipher_df.**

Output: bitstring *requested_bits*: the result of performing the **Block_Cipher_df.**

Process:

1. L = **len** (*input_string*) / 8.

NOTE 1 L is the bitstring 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 bitstring 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*. If necessary, pad S with zeroes.

4. While (**len** (S) mod *outlen* $\neq$ 0) $S = S \,||\, 0x00$.

5. *temp* = the *Null* string.

6. $i = 0$.

NOTE 4 i is represented as a 32-bit integer, i.e., **len**(i) = 32.

7. K = Leftmost *keylen* bits of 0x00010203 … 1F.

8. While (**len** (*temp*) < *keylen* + *outlen*) do:

   8.1 $IV = i \,||\, 0^{outlen - \mathbf{len}(i)}$.

   NOTE 5 This step pads the integer representation of i is padded with zeroes to *outlen* bits.

   8.2 *temp* = *temp* || **CBC_MAC** (K,(IV || S)).

   8.3 $i = i + 1$.

9. K = Leftmost *keylen* bits of *temp*.

10. X = Next *outlen* bits of *temp*.

11. *temp* = the *Null* string.

12. While (**len** (*temp*) < *no_of_bits*) do:

    12.1 X = **Block_Cipher** (K, X).

12.2 *temp* = *temp* || *X*.

13. *requested_bits* = Leftmost *no_of_bits* of *temp*.

14. **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 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 shall be used to derive the requested number of bits.

Input:

1. bitstring *Key*: The key to be used for the block cipher operation.

2. bitstring *data_to_MAC*: The data to be operated upon.

Output: bitstring *output_block*: The result to be returned from the **CBC_MAC** operation.

Process:

1. *chaining_value* =0 ^{outlen}.

NOTEThis 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 shall be 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 a *state* 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* = *blocklen* + *keylen*.

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

NOTE 1 If an implementation does not handle *additional_input*, then the *additional_input* parameter of the input may 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*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_entropy*, *max_length*).

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

   6.4 *seed_material* = *entropy_input* || *additional_input*.

   NOTE 2 If an implementation does not handle *additional_input*, then step 6.4 may be omitted, and step 6.5 may be changed to: *seed_material* = **Block_Cipher_df** (*entropy_input*, *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 to be 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* = *entropy_input* ⊕ *additional_input*.

   NOTE 4 If an implementation does not handle *additional_input*, then steps 7.3 and 7.4 may be omitted, and step 7.1 may 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 shall be 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*, bitstring *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* = *blocklen* + *keylen*.

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

NOTE 1 If an implementation will never provide *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* may be omitted as an input parameter, and step 8 may 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 will never provide *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 may or may not be 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 will never provide *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 to be used.

    If (*additional_input* $\neq$ *Null*), then:

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

    12.2 If (*temp* < *seedlen*), then *additional_input* = *additional_input* || $0^{seedlen - temp}$.

    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 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 $V = (V + 1) \bmod 2^{blocklen}$.

    14.2 *output_block* = **Block_Cipher** (*Key*, V).

    14.3 *temp* = *temp* || *output_block*.

15. *pseudorandom_bits* = Leftmost *requested_no_of_bits* of *temp*.

16. (*Key*, V) = **Update** (*additional_input*, *keylen*, *Key*, V).

NOTE 10 Update performed for backward secrecy.

NOTE 11 If an implementation will never provide *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 an ISO/IEC block cipher algorithm in the output feedback mode and shall be as specified in ISO/IEC 10116. The same block cipher algorithm and key length shall be used for all block cipher operations. The block cipher algorithm and key size shall 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* with the appropriate amount of entropy. The entropy input is used to derive a *seed*, which is then used to derive elements of the initial *state* of the DRBG. The *state* consist 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 shall be *seedlen* bits in length. The following or an equivalent process shall be used as the **Update (…)** function.

Update (…):

Input: integer *keylen*, bitstring (*seed_material*, *Key*, V).

Output: bitstring (*Key*, V).

Process:

1. *seedlen* = *blocklen* + *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 (…)

This process is the same as the generation process for **CTR_DRBG (…)** in C.3.2.2.7, except that step 14 shall be as follows:

14. While (**len** (*temp*) < *requested_no_of_bits*) do:

    14.1. V = **Block_Cipher** (*Key*, V).

    14.2. *temp* = *temp* || V.

C.4 DRBGs based on number theoretic problems

C.4.1 Introduction to DRBGs based on number theoretic problems

A DRBG can be designed to take advantage of number theoretic problems (e.g., the discrete logarithm problem). If done correctly, such a generator's properties of randomness and/or unpredictability will be assured by the difficulty of finding a solution to that problem. C.4.2 specifies a DRBG based on elliptic curves. C.4.3 specifies a DRBG related to the well known RSA problem.

C.4.2 Dual Elliptic Curve DRBG (Dual_EC_DRBG)

C.4.2.1 Discussion

Dual_EC_DRBG (…) is based on the following hard problem, sometimes known as the "elliptic curve discrete logarithm problem," given points P and Q on an elliptic curve of order n, find a such that $Q = aP$.

Dual_EC_DRBG (…) uses a seed m bits in length to initiate the generation of *blocksize*-bit pseudorandom strings by performing scalar multiplications on two points in an elliptic curve group, where the curve is defined over a field approximately 2^m in size. *blocksize* has been chosen to ensure full entropy in the output strings; it is a multiple of 8 close to but no larger than m -16. (B.2.2 has details.) Selecting an m as small as possible, subject to the security strength required by the application, may result in improved performance. For all the curves provided in D.1, $m \geq 163$.

The instantiation of this DRBG requires the selection of an appropriate elliptic curve and curve points which are specified in D.1 for the desired security strength.

The *seed* used to define the initial value (S) of the DRBG shall have entropy that is at least the maximum of 128 and the desired security *strength* (i.e., entropy ≥ **max** (128, strength)). The seed length shall be m bits. Further requirements for the *seed* are provided in 9.3.

Backward secrecy is inherent in the algorithm, even if the internal state is compromised. Forward secrecy is also inherent when observed from outside the DRBG boundary. If an application is concerned about compromise of the hidden state in an instantiation of the **Dual_EC_DRBG (…)**, the state may be infused with new entropy in a number of ways, as discussed in C.4.2.2.5.

When optional additional input (*additional_input*) is used, the value of *additional_input* is arbitrary, and it will be hashed to an m-bit string.

C.4.2.2 Description

C.4.2.2.1 General

The instantiation of **Dual_EC_DRBG (…)** consists of selecting an appropriate elliptic curve and point pairing from D.1 and obtaining a *seed* that is used to determine an initial value (S) for the DRBG that is one element of the initial *state*. The state consists of:

1. a counter (*reseed_counter*) that indicates the number of blocks of random bits produced by the **Dual_EC_DRBG (…)** during the current instance and since the previous reseeding;
2. a value (S) which determines the current position on *E*. S is updated during each request for pseudorandom bits;
3. the elliptic curve domain parameters (*curve_type*, m, p, a, b, n), where *curve_type* indicates a prime field F_p, or a pseudorandom or Koblitz curve over the binary field $F_2{}^m$; a and b are two field elements that define the equation of the curve, and n is the order of the point G; unless a binary curve type is requested at initialisation the default *curve_type* 0 indicating mod p, will be used;
4. two points P and Q on the curve; the generating point G specified for the chosen curve will be used as P;
5. the security *strength* provided by the instance of the DRBG; the curve will be selected to provide a minimum of *requested_strength* bits of security;
6. a *prediction_resistance_flag* that indicates whether or not forward secrecy is required by the DRBG; setting this flag forces a call to the reseed function each time that the **Dual_EC_DRBG (…)** is invoked for a random bitstring; and
7. a record of the seeding material in the form of a one-way function that is performed on the *seed* for later comparison with a new *seed* when the DRBG is reseeded.

The variables used in the description of **Dual_EC_DRBG (…)** are:

a, b	Two field elements that define the equation of the curve.
additional_input	The hashed bitstring derived from the optional *additional_input_string*.
additional_input_string	Optional additional input. A byte array that may be provided on any call for random bits or during reseeding. The string will be hashed to m bits using **Hash_df (…)**.
blocksize	The number of bits output by a single step of the **Dual_EC_DRBG (…).** The precise value depends on the curve chosen, but is always a multiple of 8 near m-16. (See B.2 and C.4.2.2.2.)
curve_type	Either 0 (*Prime_field_curve*), 1 (*Random_binary_curve*), or 2 (*Koblitz_curve*) indicating a curve over a prime field, a random binary curve, or a Koblitz curve, respectively. The default curve type is 0 (i.e., mod *p* will be used).
E	An elliptic curve defined over F_p or $F_2{}^m$.
entropy_input	The bits containing entropy that are used to determine *seed_material* and generate a seed.

F	The cofactor of the curve: 1 for all prime field curves, 2 or 4 for the binary curves. NOTE 1 This value will be implicit from the *curve_type* and a.
G	A generating point of prime order n on the curve *E*.
Get_entropy (*min_entropy*, *min_length*, *max_length*)	A function that acquires a string of bits from an entropy 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 (*hash_input***)**	An ISO/IEC approved hash-function that shall be as specified in ISO/IEC 10118-3, returning a bitstring whose input *hash_input* may be any multiple of 8 bits in length.
Hash_df (*hash_input*, *output_len***)**	A function to distribute the entropy in *hash_input* to a bitstring *output_len* long. The function **Hash (…)** is used to do this. *hash_input* may be any multiple of 8 bits in length; *output_len* is arbitrary.
i	A temporary value that is used as a loop counter.
m	Length in bits of the internal state S; the curve is defined over a field with approximately 2^m elements.
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_length* is a multiple of 8 bits.
min_entropy	The minimum amount of entropy to be obtained from the entropy source and provided in the seed. NOTE 2 In fact, the value of *strength* is used in this determination, and *strength* is always at least *requested_strength*.
min_length	The minimum length of the *entropy_input*.
n	The order of the generating point G on the curve.
Null	The null (i.e., empty) string.
p	The modulus when *curve_type* = 0 (prime field); an m-bit prime.
P, Q	Two points on the elliptic curve *E*, such that each generates a large cyclic subgroup on *E*. The generating point G will be used as P.
pad8 (*bitstring*)	A function that inputs an arbitrary length bitstring and returns a copy of that bitstring padded on the right with binary 0's, if necessary, to a multiple of 8. NOTE 3 This is an implementation convenience for byte-oriented functions.
personalisation_string	A byte array that can provide additional assurance of seed uniqueness at instantiation.
prediction_resistance_flag	An instantiation flag indicating whether or not forward secrecy is to be provided by the DRBG. By default, this flag is not set. Setting this flag to 1 forces a call to **Reseed_Dual_EC_DRBG_Instantiation (…)** from within **Dual_EC_DRBG (…)** each time the process is invoked.

pseudorandom_bits	The pseudorandom bits produced by the DRBG.
R	A value from which pseudorandom bits are extracted.
requested_curve_type	The *curve_type* can be specified as input to **Instantiate_Dual_EC_DRBG (…)**; if none is requested the default value of 0 (*Prime_field_curve*) is assigned.
requested_no_of_bits	The number of pseudorandom bits to be returned from **Dual_EC_DRBG (…)** function.
requested_strength	The security strength to be associated with the requested pseudorandom bits.
reseed_counter	A count of the number of iterations of the **Dual_EC_DRBG (…)** since the last reseeding.
reseed_interval	The maximum number of steps taken along the curve before the DRBG shall be reseeded. The value 10000 shall be used (see B.2.4).
s	A temporary value.
S	A value that is initially determined by a *seed*, but assumes new values during each request of pseudorandom bits from the DRBG.
seed_material	The seed used to derive the initial value of S.
state(*state_handle*)	An array of states for different DRBG instantiations. A state is carried between DRBG calls. For the **Dual_EC_DRBG (…)**, the state for an instantiation is defined as *state*(*state_handle*) = {*reseed_counter*, S, *curve_type*, m, p, a, b, n, P, Q, *strength*, *prediction_resistance_flag* }. A particular element of the *state* is specified as *state*(*state_handle*).*element*, e.g., *state*(*state_handle*).S. NOTE 4 p is only needed by the *curve_type* = 0 curves (*Prime_field_curve*).
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 maximum strength of an instance of the DRBG (i.e., 80, 112, 128, 192 or 256).
temp	A temporary value.
temp_input	A temporary input value.
Truncate (*bits*, *in_len*, *out_len*)	A function that inputs a bit string of *in_len* bits, returning a string consisting of the leftmost *out_len* bits of input. If *in_len* < *out_len*, the input string is padded on the right with (*out_len* - *in_len*) zeroes, and the result is returned.
$x(A)$	The x coordinate of the point A on the curve *E*.
φ	A mapping from field elements to non-negative integers, which takes the bit vector representation of a field element and interprets it as the binary expansion of an integer.
$*$	Scalar multiplication of a point on the curve.

C.4.2.2.2 Instantiation of Dual_EC_DRBG (…)

The following process or its equivalent shall be used to instantiate the **Dual_EC_DRBG (…)** process. Let **Hash (…)** be an approved hash-function for the security strengths to be supported. If the DRBG will be used for multiple security strengths, and only a single hash-function will be available, that hash-function shall be suitable for all supported security strengths.

Instantiate_Dual_EC_DRBG (…):

Input: integer (*requested_strength*, *prediction_resistance_flag*, *requested_curve_type*), string *personalisation_string*.

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

Process:

1. If (*requested_strength* > 256), then **Return** (Failure message).

2. If (*requested_curve_type* = *Prime_field_curve*), then

   If (*requested_strength* ≤ 80), then {*strength* = 80, m = 192}

   NOTE 1 There is no recommended curve with m = 160. The smallest mod p curve recommended in D.1 is for m = 192. Therefore, when the DRBG is instantiated with a nominal strength of 80, the actual strength is 96.

   Else if (*requested_strength* ≤ 112), then {*strength* = 112, m = 224}

   Else if (*requested_strength* ≤ 128), then {*strength* = 128, m = 256}

   Else if (*requested_strength* ≤ 192), then {*strength* = 192, m = 384}

   Else if (*requested_strength* ≤ 256), then {*strength* = 256, m = 521}.

   NOTE 2 There is no recommended curve with m = 512.

3. If (*requested_curve_type* ≠ *Prime_field_curve*), then

   If (*requested_strength* ≤ 80), then {*strength* = 80, m = 163}

   Else if (*requested_strength* ≤ 112), then {*strength* = 112, m = 233}

   Else if (*requested_strength* ≤ 128), then {*strength* = 128, m = 283}

   Else if (*requested_strength* ≤ 192), then {*strength* = 192, m = 409}

   Else if (*requested_strength* ≤ 256), then {*strength* = 256, m = 571}.

4. Choose a suitable elliptic curve *E* defined over the prime field Fp or the binary field F_2^m ,where p is an m-bit prime, from D.1.

   If (*curve_type* = *Prime_field_curve*), then select elliptic curve P–m.

   Else if (*curve_type* = *Random_binary_curve*), then select elliptic curve B–m and set p = 0.

   Else if (*curve_type* = *Koblitz_curve*), then select elliptic curve K–m and set p = 0.

5. Set the point P to the generator G and n to the order of G.

6. Set the corresponding point Q from D.1.

7. Set the *blocksize*—the number of bits to use on each iteration of the **Dual_EC_DRBG (…).** As explained in B.2 this number depends on the *curve_type* and its size m. Only the rightmost *blocksize* bits of each block produced are output; the others are discarded. The formula for *blocksize* is [largest multiple of 8 smaller than m - (13+$\log_2(f)$)]. The following table summarizes the *blocksize* calculation, in the format [ISO/IEC curve: *blocksize*]:

Table C.3 — Elliptic curve block sizes

Prime_field_curve	***Random_binary_curve***	***Koblitz_curve***
P-192 : 176	B-163 : 144	K-163 : 144
P-224 : 208	B-233 : 216	K-233 : 216
P-256 : 240	B-283 : 264	K-283 : 264
P-384 : 368	B-409 : 392	K-409 : 392
P-521 : 504	B-571 : 552	K-571 : 552

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

9. *min_length* = 8$\lceil m/8 \rceil$.

10. (*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_length*, *max_length*).

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

12. *seed_material* = *entropy_input* || *personalisation_string*.

13. Use a hash-function to ensure that the entropy is distributed throughout the bits: S = **Hash_df** (*seed_material*, m).

14. Initialise *reseed_counter* to 0.

NOTE 3 *reseed_counter* is incremented every *blocksize* bits.

15. Save all curve and state information:

state(*state_handle*) = {*reseed_counter*, S, *curve_type*, m, p, a, b, n, P, Q, *strength*, *prediction_resistance_flag* }.

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

C.4.2.2.3 Reseeding Dual_EC_DRBG (…) Instantiation

The following process or its equivalent shall be used to reseed the **Dual_EC_DRBG (…)** process, after it has been instantiated.

Reseed_Dual_EC_DRBG_Instantiation (…):

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

Output: string *status*.

Process:

1. If a *state* is not available, **Return** (Failure message).
2. Get the appropriate *state* values, e.g., S = *state*(*state_handle*).S, m = *state*(*state_handle*).m, *strength* = *state*(*state_handle*).*strength*.
3. *min_entropy* = **max** (128, *strength*).
4. *min_length* = 8$\lceil m/8 \rceil$.
5. (*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_length*, *max_length*).
6. If (*status* $\neq$ "Success"), then **Return** (Failure message).
7. Combine new entropy_input with the old state and any *additional_input*:

   seed_material = **pad8** (S) || *entropy_input* || *additional_input_string*.
8. S = **Hash_df** (*seed_material*, m).
9. Update the appropriate *state* values.

   9.1 *state*(*state_handle*).S = S.

   9.2 *state*(*state_handle*).*reseed_counter* = 0.
10. **Return** ("Success").

C.4.2.2.4 Generating pseudorandom bits using Dual_EC_DRBG (…)

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

Dual_EC_DRBG (…):

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

Output: string *status*, bitstring *pseudorandom_bits*.

Process:

1. If a *state* is not available, **Return** (Failure message, *Null*).
2. Get the appropriate *state* values, e.g., S = *state*(*state_handle*).S, m = *state*(*state_handle*).m, *strength* = *state*(*state_handle*).*strength*, P = *state*(*state_handle*).P, Q = *state*(*state_handle*).Q, *reseed_counter* = *state*(*state_handle*).*reseed_counter*, *prediction_resistance_flag* = *state*(*state_handle*).*prediction_resistance_flag*.

3. Check that the *requested_strength* is not more than that provided by this instantiation.

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

4. Check for a request to supply additional input. This will be added to the state on the first iteration only.

   If (*additional_input_string* = *Null*), then *additional_input* = 0

   Else: *additional_input* = **Hash_df** (**pad8**(*additional_input_string*), m).

   NOTE 1 *additional_input* set to m zeroes.

   NOTE 2 hash to m bits.

5. If the prediction resistance flag has been set, instil new entropy with a call to reseed the **Dual_EC_DRBG (…)**. **Reseed_Dual_EC_DRBG_Instantiation (…)** resets *reseed_counter* = 0.

   If (*prediction_resistance_flag* = 1), then

   5.1 *status* = **Reseed_Dual_EC_DRBG_Instantiation** (*state_handle*, *Null*).

   5.2 If (*status* ≠ "Success"), then **Return** (*status*, *Null*).

   5.3 S = *state*(*state_handle*).S, *reseed_counter* = *state*(*state_handle*).*reseed_counter*.

6. *temp* = the *Null* string; i = 0.

7. Determine if reseeding is required:

   NOTE 3 Following steps produce *requested_no_of_bits* , *blocksize* at a time.

   If (*reseed_counter* = 10000), then:

   7.1 *status* = **Reseed_Dual_EC_DRBG_Instantiation** (*state_handle*, *Null*).

   7.2 If (*status* ≠ "Success"), then **Return** (*status*, *Null*).

   7.3 S = *state*(*state_handle*).S, *reseed_counter* = *state*(*state_handle*).*reseed_counter*.

8. s = $S \oplus$ *additional_input*.

NOTE 4 s is to be interpreted as an m-bit unsigned integer. To be precise, when *curve_type* = *Prime_field_curve*, s should be reduced mod n; the operation $*$ will effect this.

9. $S = \varphi(x(s * P))$.

NOTE 5 S is an m-bit number. See footnote [2].

10. $R = \varphi(x(S * Q))$.

NOTE 6 R is an m-bit number. [2)]

11. *temp* = *temp* || (rightmost *blocksize* bits of R).

12. *additional_input* = 0.

NOTE 7 m zeroes. *additional_input_string* is added only on the first iteration.

13. *reseed_counter* = *reseed_counter* + 1.

14. $i = i + 1$.

15. If (|*temp*| < *requested_no_of_bits*), then go to step 7.

16. *pseudorandom_bits* = **Truncate** (*temp*, $i \times$ *blocksize*, *requested_no_of_bits*).

17. $S = \varphi(x(S * P))$.

18. Update the changed values in the *state*.

    18.1. *state*(*state_handle*).$S = S$.

    18.2. *state*(*state_handle*).*reseed_counter* = *reseed_counter*.

19. **Return** ("Success", *pseudorandom_bits*).

C.4.2.2.5 Inserting additional entropy into the state of Dual_EC_DRBG (…)

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

1. calling the **Reseed_Dual_EC_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. utilising the automatic reseeding feature of the **Dual_EC_DRBG (…)**. The automatic reseeding feature will force a call to **Reseed_Dual_EC_DRBG_Instantiation (…)** for new entropy whenever 10000 blocks of random have been output since the last call to reseed;

3. setting *prediction_resistance_flag* = 1 at instantiation. This forces a call to **Reseed_Dual_EC_DRBG_Instantiation (…)** each time that **Dual_EC_DRBG(…)** is invoked; or

---

[2)] The precise definition of $\varphi(x)$ used in steps 9 and 10 depends on the field representation of the curve points. In keeping with the convention of FIPS 186-3, the following elements will be associated with each other:

B: $| c_{m-1} | c_{m-2} | \ldots | c_1 | c_0 |$, a bitstring, with c_{m-1} being leftmost ;

Z: $c_{m-1}2^{m-1} + \ldots + c_2 2^2 + c_1 2^1 + c_0 \in Z$;

Fa: $c_{m-1}2^{m-1} + \ldots + c_2 2^2 + c_1 2^1 + c_0 \bmod p \in \mathrm{GF}(p)$;

Fb: $c_{m-1}t^{m-1} \oplus \ldots \oplus c_2 t^2 \oplus c_1 t \oplus c_0 \in \mathrm{GF}(2^m)$, when a polynomial basis is used;

Fc: $c_{m-1}\beta \oplus c_{m-2}\beta^2 \oplus c_{m-3}\beta^{2^2} \oplus \ldots \oplus c_0\beta^{2^{m-1}} \in \mathrm{GF}(2^m)$, when a normal basis is used.

Thus, any field element *x* of the form *Fa*, *Fb* or *Fc* will be converted to the integer *Z* or bitstring *B*, and vice versa, as appropriate.

4. supplying additional input during any call to **Dual_EC_DRBG (…)** for random bits.

NOTE Frequent calls to the **Get_entropy (…)** function may cause severe performance degradation with this or any DRBG.

C.4.3 Micali Schnorr DRBG (MS_DRBG)

C.4.3.1 Discussion

The **MS_DRBG (…)** is a variant of the so-called RSA generator, which is defined as follows.

Let gcd(x, y) denote the greatest common divisor of integers x and y, and $\phi(n)$ represents the Euler phi function. Select n, the product of two distinct large primes; and e, a positive integer such that gcd(e, $\phi(n)$) = 1. Define $f(y) = y^e \bmod n$. Starting with a seed y_0, form the sequence $y_{i+1} = f(y_i)$, and output the string consisting of the k = lg lg(n) least significant bits of each y_i. These bits are (asymptotically in n) known to be as secure as the RSA function f.

The Micali-Schnorr generator **MS_DRBG (…)** uses the same e and n to produce many more random bits per iteration, while eliminating the reuse of bits as both output and seed. Each $y_i \in [0, n)$ is viewed as the concatenation $s_i \| z_i$ of an r-bit number s_i and a k-bit number z_i (where k = lg(n)-r). The s_i is used to propagate the integer sequence: $y_{i+1} = s_i^e \bmod n$; the z_i are output as random bits. Note that r shall be at least 2(min{*strength*, lg(n)/e}), where *strength* is the desired security strength of the generator, and $e \geq 3$. A random r-bit *seed* s_0 is used to initialise the process.

The **MS_DRBG (…)** is cryptographically secure under the stronger assumption that sequences of the form $s^e \bmod n$ are "the same" as sequences of random integers in Z_n. Here the s are assumed to be r-bit integers, and "the same" means indistinguishable by any polynomial-time algorithm. Accepting the stronger assumption allows for k to be a significant percentage of lg(n).

The lengths r and k, the RSA modulus n, and the value of exponent e are variable within the bounds described below. The bounds are based on the desired *strength* of bits produced. For maximum efficiency e should be kept small and k large. The k bits generated at each step are concatenated to form pseudorandom bit strings of any desired length.

Seeding material is provided by the implementation-dependent function **Get_entropy (…)**. The minimum entropy required from this function will be set to **max** (128, *strength*).

Backward secrecy is inherent in the algorithm, even if the internal state is compromised. Forward secrecy is also inherent when observed from outside the DRBG boundary. If an application is concerned about compromise of the hidden state in an instantiation of the **MS_DRBG (…)**, the state may be infused with new entropy in a number of ways.

When optional additional input (*additional_input*) is used, the value of *additional_input* is arbitrary and it will be hashed to an r-bit string.

C.4.3.2 Description

C.4.3.2.1 General

At initialisation of **MS_DRBG (…)** the MS parameters n, e, r, and k are selected as described below, and a random initial *seed* s_0 is obtained. Each of these become part of the internal *state* of the DRBG. The state consists of:

1. the MS parameters n, e, r and k;

2. a integer $S \in [0,2^r)$ that propagates the internal state sequence from which pseudorandom bits are derived;

3. the security *strength* provided by the instance of the DRBG. For efficiency, the smallest modulus size lg(n) providing *requested_strength* bits of security will be selected from Table C.4. The actual *strength* provided is stored in *state*;

4. the mininum entropy needed from a call to **Get_entropy (…)** for seeding material. The value of *min_entropy* will be set to **max** (128, *strength*);

5. a counter (*reseed_counter*) that indicates the number of blocks of random produced by **MS_DRBG (…)** during the current instance and since the previous reseeding;

6. a *prediction_resistance_flag* that indicates whether or not forward secrecy is required by the DRBG; setting this flag forces a call to the reseed function each time **MS_DRBG (…)** is invoked for a random bitstring; and

7. a record of the seeding material in the form of a one-way **Hash (…)** function that is performed on the *seed* for later comparison with a new *seed* when the DRBG is reseeded;

The variables used in the description of **MS_DRBG (…)** are:

additional_input	The hashed bitstring derived from the optional *additional_input_string*.
additional_input_string	Optional additional input. A byte array that may be provided on any call for random bits. The string will be hashed to *r* bits using **Hash_df (…)**.
e	A positive integer used as an RSA exponent.
entropy_input	The bits containing entropy that are used to determine *seed_material* and generate a *seed*.
gcd(x,y)	The greatest common divisor of the integers x and y.
Get_entropy (*min_entropy*, *min_length*, *max_length*)	A function that acquires a string of bits from an entropy 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. The **MS_DRBG** will always specify *min_length* = *max_length* = r.
Get_random_modulus (lg(n), e)	A function that generates a random RSA modulus n of size lg(n) bits, which satisfies **gcd** (e, $\phi(n)$) = 1, using an approved algorithm. (See C.4.3.2.2.)
Hash (*hash_input*)	An ISO/IEC approved hash-function that shall be as specified in ISO/IEC 10118-3, returning a bitstring whose input *hash_input* may be any multiple of 8 bits in length.
Hash_df (*hash_input*, *output_len*)	A function to distribute the entropy in *hash_input* to a bitstring *output_len* long. The function **Hash (…)** is used to do this. *hash_input* may be any multiple of 8 bits in length; *output_len* is arbitrary.
i	A temporary value that is used as a loop counter
k	Number of bits generated at each iteration of **MS_DRBG**. As an implementation convenience this will always be a multiple of 8 bits.

lg(n)	Number of bits in the binary representation of n.
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_length* is a multiple of 8 bits.
min_entropy	The minimum amount of entropy to be obtained from the entropy source and provided in the seed. NOTE 1 In fact the value of *strength* is used in this determination, and *strength* is always at least *requested_strength*.
min_length	The minimum length of the *entropy_input*.
M-S parameters	n, e, r, k
n	The RSA modulus; the product of two distinct large primes p, q.
Null	The null (i.e., empty) string.
old_transformed_entropy_input	A record of the *entropy_input* used in this previous instance of the DRBG.
p, q	Prime numbers generated using an approved algorithm. These prime numbers shall be as specified in ISO/IEC 18032. These will be randomly generated at initialisation if *use_random_primes* is set to 1. Otherwise the default modulus of an appropriate size will be used.
pad8(*bitstring*)	A function which inputs an arbitrary length bitstring and returns a copy of that bitstring padded on the right with binary 0's, if necessary, to a multiple of 8. NOTE 2 This is an implementation convenience for byte-oriented functions.
personalisation_string	A byte array that can provide additional assurance of seed uniqueness at instantiation.
prediction_resistance_flag	An instantiation flag indicating whether or not forward secrecy is to be provided by the DRBG. By default, this flag is not set. Setting this flag to 1 causes a reseeding to be done on every invocation of **MS_DRBG (…)**.
pseudorandom_bits	The pseudorandom bits produced by the DRBG.
r	Bit length of the seeds s_i; r = lg(n) - k. NOTE 3 r will always be a multiple of 8 bits.
R	A value from which pseudorandom bits are extracted.
requested_e	Requested RSA exponent e.
requested_k	Requested size k of each output string.
requested_no_of_bits	The number of pseudorandom bits to be returned from **MS_DRBG (…)** function.

requested_strength	The security strength to be associated with the requested pseudorandom bits.
reseed_counter	A count of the number of iterations of the **MS_DRBG (…)** since the last reseeding.
reseed_interval	The maximum number of blocks of random output produced before the DRBG shall be reseeded. The value of 50000 shall be used.
S	A value that is initially determined by a seed, but assumes new values during each request of pseudorandom bits from the DRBG.
seed_material	The seed used to derive the initial value of S.
state(*state_handle*)	An array of states for different DRBG instantiations. A state is carried between DRBG calls. For the **MS_DRBG (…)**, the state for an instantiation is defined as *state*(*state_handle*) = { *reseed_counter*, S, n, e, r, k, *strength*, *prediction_resistance_flag*, *transformed_entropy_input* }. A particular element of the *state* is specified as *state*(*state_handle*).*element*, e.g., *state*(*state_handle*).S
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* of the bits requested from the DRBG. It will always be at least *requested_strength*.
transformed_entropy_input	A record of the *seed_material* used in the current instance of the DRBG.
Truncate (*bits*, *in_len*, *out_len*)	A function which inputs a bit string of *in_len* bits, returning a string consisting of the leftmost *out_len* bits of input. If *in_len* < *out_len* the input string is returned padded on the right with *out_len* – *in_len* zeroes.
use_random_primes	If set to 1 (*Use_random_primes*), random primes of size lg(n)/2 will be generated at initialisation, using an approved algorithm, and having entropy at least *min_entropy*. If set to 0 (*Random_primes_not_required*), the appropriate modulus from D.2 shall be used.
y_i	An integer, $y_i \in [0, n)$. $y_i = s_i \,||\, z_i$.
z_i	k-bit output of **MS_DRBG(…)** at each iteration i.
$\phi(n)$	The Euler phi function: $\phi(n)$ = the number of positive integers < n which are relatively prime to n. For an RSA modulus $n = pq$, $\phi(n) = (p\text{-}1)(q\text{-}1)$.

C.4.3.2.2 Selection of the MS parameters

The instantiation of **MS_DRBG (…)** consists of selecting an appropriate RSA modulus n and exponent e; sizes r and k for the seeds and output strings, respectively; and a starting seed.

The MS parameters n, r, e and k are selected to satisfy the following six conditions, based on *strength*:

1.	$1 < e < \phi(n)$; **gcd** $(e, \phi(n)) = 1$;	ensures that the mapping $s \rightarrow s^e$ mod n is 1 to 1
2.	$re \geq 2\lg(n)$;	ensures that the exponentiation requires a full modular reduction
3.	$r \geq 2$*strength* ;	protects against a table attack
4.	k, r are multiples of 8;	an implementation convenience
5.	$k \geq 8$; $r + k = \lg(n)$;	all bits are used
6.	$n = pq$	strong (as in ISO/IEC 18032), secret primes

The MS parameters are determined in this order:

1. The size of the modulus $\lg(n)$ is set first. It shall conform to the values given in Table C.4 for the requested security *strength*.

Table C.4 — Equivalent security strengths

Bits of Security	RSA
80	$\lg(n) = 1024$
112	$\lg(n) = 2048$
128	$\lg(n) = 3072$
192	$\lg(n) = 7680$
256	$\lg(n) = 15360$

2. The RSA exponent e. The implementation should allow the application to request any *odd* integer e in the range $1 < e < 2^{\lg(n)-1} - 2^{*}2^{\frac{1}{2}\lg(n)}$.

   NOTE 1 The inequality ensures that $e < \phi(n)$ when an approved algorithm is used to generate the primes p, q.

   If no exponent is requested the default value $e = 3$ should be used.

3. The number k of output bits used for each iteration. The implementation should allow any multiple of 8 in the range $8 \leq k \leq \min\{\lg(n) - 2$*strength*, $\lg(n) - 2\lg(n)/e\}$. If not specified k should be selected as the *largest* multiple of 8 in the allowable range.

   Any values for *requested_e* and *requested_k* outside these ranges shall be flagged an error.

4. Set the size r of the seeds: $r = \lg(n) - k$

5. Selection of the modulus n. The application may request a private modulus, or it may use the default modulus of the appropriate size (as given in D.2). The implementation shall permit either, based on the value of *use_random_primes*.

If *use_random_primes* = 1, two primes p and q of size lg(n)/2 bits , having entropy at least *min_entropy* , and satisfying **gcd** $(e, (p\text{-}1)(q\text{-}1)) = 1$ shall be generated, using an approved algorithm. Suitable algorithms are specified in ISO/IEC 18032. An implementation shall use strong primes as defined in that standard: each of p-1, p+1, q-1 and q+1 shall have a large prime factor of at least *strength* bits.

NOTE 2 Any approved algorithm will generate a modulus of size lg(n) bits using strong primes of size lg(n)/2 bits, and will allow the exponent e to be specified beforehand.

The difficulty of the RSA problem relies on the secrecy of the primes p and q comprising the modulus. Whenever private primes are generated the implementation shall clear memory of those values prior to leaving the initialisation routine. Only the modulus n shall be kept in the internal *state*.

If *use_random_primes* = 0 the appropriate modulus from D.2 shall be used.

These moduli have been generated using strong primes of the form $p = 2p_1+1$, $q = 2q_1+1$, where p_1 and q_1 are themselves prime. In addition p+1 and q+1 each have the required large prime factor.

NOTE 3 This choice of strong primes essentially guarantees that any odd exponent e in the allowable range that might be requested will be relatively prime to $\phi(n)$.

C.4.3.2.3 Instantiation of MS_DRBG(…)

The following process or its equivalent shall be used to initialise the **MS_DRBG (…)** process. Let **Hash (…)** be an approved hash-function for the security strengths to be supported. If the DRBG will be used for multiple security strengths, and only a single hash-function will be available, that hash-function shall be suitable for all supported security strengths.

Instantiate_MS_DRBG (…):

Input: integer (*requested_strength*, *prediction_resistance_flag*, *use_random_primes*, *requested_e*, *requested_k*), string *personalisation_string*.

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

Process:

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

2. Determine modulus size lg(n) appropriate for the requested strength using Table C.4

   If (*requested_strength* ≤ 80) then *strength* = 80, lg(n) = 1024

   Else if (*requested_strength* ≤ 112) then *strength* = 112, lg(n) = 2048

   Else: if (*requested_strength* ≤ 128) then *strength* = 128, lg(n) = 3072

   Else: if (*requested_strength* ≤ 192) then *strength* = 192, lg(n) = 7680

   Else: if (*requested_strength* ≤ 256) then *strength* = 256, lg(n) = 15360

   Else: **Return** (Failure message).

3. Select the exponent size e. The default size is e = 3.

   If (*requested_e* = 0), then e = 3

Else: Check bounds

{

If (e < 3), then **Return** (Failure message);

NOTE 1 The integer e is at least 3.

If ($e \geq 2^{\lg(n)-1} - 2^{\lg(n)/2+1}$), then **Return** (Failure message);

NOTE 2 This step ensures that e will need to be less than $\phi(n)$.

If (e is even), then **Return** (Failure message);

NOTE 3 e will need to be relatively prime to $\phi(n)$, hence odd.

}

4. Select the output length k. The **MS_DRBG (…)** uses the least significant k bits of $y_i = s_i \,||\, z_i$ on each iteration. The default size is to use the largest possible.
If (*requested_k* = 0), then:

k = min {$\lfloor \lg(n) - 2strength \rfloor$, $\lfloor \lg(n)(1 - 2/e) \rfloor$}

NOTE 4 $3 \leq e < 2^{\lg(n)-1} - 2(2^{\frac{1}{2}\lg(n)}) \Rightarrow 8 \leq 2/3 \lg(n) \leq \lfloor \lg(n)(1 - 2/e) \rfloor \leq \lg(n) - 1$

Round down to a multiple of 8:

$k = 8\lfloor k/8 \rfloor$

Else:

{

NOTE 5 The following checks the bounds.

k = *requested_k*

If (k < 1), then **Return** (Failure message)

If (k > min {$\lfloor \lg(n) - 2strength \rfloor$, $\lfloor \lg(n)(1 - 2/e) \rfloor$}),

then **Return** (Failure message)

If (k is not a multiple of 8),

then **Return** (Failure message)

}

5. Set the size of the seeds

$r = \lg(n) - k$

NOTE 6 $r \geq 2strength$

6. Select the modulus n. *use_random_primes* determines whether the default values are used or a private modulus is generated.

If (*use_random_primes* = *Random_primes_not_required*)

then Set n based on the size lg(n) from the list in D.2

Else
{

(*status*, n) = **Get_random_modulus** (lg (n), e)

If (*status* $\neq$ "Success"), then **Return** (Failure message)

}

NOTE 7 An approved function is used to generate a random modulus n of the appropriate size, having strong primes as factors, and for which **gcd** ($\phi(n)$,e) = 1.

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

8. *min_length* = r.

9. (*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_length*, *max_length*).

10. If (*status* $\neq$ "Success"), then **Return** (Failure message).

11. *seed_material* = *entropy_input* || *personalisation_string*.

12. Use a hash-function to ensure that the entropy is distributed throughout the bits: S = **Hash_df** (*seed_material*, r).

13. Perform a one-way function on the seed material for later comparison: *transformed_entropy_input* = **Hash** (*entropy_input*).

14. *reseed_counter* = 0.

NOTE 8 *reseed_counter* is incremented every k bits.

15. Store all values in *state*.

state(*state_handle*) = {*reseed_counter*, S, n, e, r, k, *strength*, *prediction_resistance_flag*, *transformed_entropy_input* }.

16. **Return** ("Success").

C.4.3.2.4 Reseeding MS_DRBG (…) Instantiation

The following process or its equivalent shall be used to reseed the **MS_DRBG (…)** process, after it has been instantiated.

Reseed_MS_DRBG_Instantiation (…):

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

Output: string *status*.

Process:

1. If a *state* is not available, then **Return** (Failure message).

2. Get the appropriate *state* values, e.g., S = *state*(*state_handle*).S, r = *state*(*state_handle*).r, *old_transformed_entropy_input* = *state*(*state_handle*).*transformed_entropy_input*.

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

4. *min_length* = r.

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

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

7. Perform a one-way function on the seed material for comparison: *transformed_entropy_input* = **Hash** (*entropy_input*).

8. Check for a viable entropy source:

   If (*transformed_entropy_input* = *old_transformed_entropy_input*),

   then **Return** (Failure message).

9. Combine new *entropy_input* with old state and any *additional_input*:

   seed_material = S || *entropy_input* || *additional_input_string*.

10. S = **Hash_df** (*seed_material*, r).

11. Update the appropriate *state* values, and reset the *reseed_counter* to 0.

    11.1 *state*(*state_handle*).S =S.

    11.2 *state*(*state_handle*).*transformed_entropy_input* = *transformed_entropy_input*.

    11.3 *state*(*state_handle*).*reseed_counter* = 0.

12. **Return** ("Success").

C.4.3.2.5 Generating pseudorandom bits using MS_DRBG (…)

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

MS_DRBG (…):

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

Output: string *status*, bitstring *pseudorandom_bits*.

Process:

1. If no *state* exists, then **Return** (Failure message, *Null*).

2. Get the appropriate *state* values, e.g., S = *state*(*state_handle*).S, n = *state*(*state_handle*).n, e = *state*(*state_handle*).e, k = *state*(*state_handle*).k, r = *state*(*state_handle*).r, *strength* = *state*(*state_handle*).*strength*, *reseed_counter* = *state*(*state_handle*).*reseed_counter*, *prediction_resistance_flag* = *state*(*state_handle*).*prediction_resistance_flag*.

3. Check that the requested strength is not larger than that provided by this instantiation.

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

4. Check for a request to supply additional input. This will be added to the state on the first iteration only.

   If (*additional_input_string* = *Null*), then *additional_input* = 0

   NOTE 1 *additional_input* set to r zeroes.

   Else *additional_input* = **Hash_df** (**pad8**(*additional_input_string*), r).

   NOTE 2 hash to r bits.

   NOTE 3 If a prediction resistance request has been made, the following steps will instil new entropy with a call to reseed the **MS_DRBG (…)**. The reseed process resets *reseed_counter* to 0.

5. If (*prediction_resistance_flag* = 1), then:

   5.1 *status* = **Reseed_MS_DRBG_Instantiation** (*state_handle*, *Null*).

   5.2 If (*status* ≠ "Success"), then **Return** (*status*, *Null*).

   5.3 S = *state*(*state_handle*).S, *reseed_counter* = *state*(*state_handle*).*reseed_counter*.

6. *temp* = the *Null* string; i = 0.

7. Determine if reseeding is required. **Reseed_MS_DRBG_Instantiation(…)** resets *reseed_counter* to 0

   If (*reseed_counter* = 50000), then:

   7.1 *status* = **Reseed_MS_DRBG_Instantiation** (*state_handle*, *Null*).

   7.2 If (*status* ≠ "Success"), then **Return** (*status*, *Null*).

   7.3 S = *state*(*state_handle*).S, *reseed_counter* = *state*(*state_handle*).*reseed_counter*.

8. $s = S \oplus additional\_input$.

NOTE 4 s is to be interpreted as an r-bit unsigned integer.

9. $S = (s^e \bmod n) \bmod 2^k$.

NOTE 5 S is an r-bit number.

10. $R = (s^e \bmod n) \bmod 2^k$

NOTE 6 R is a k-bit number.

11. *temp* = *temp* || R .

12. *additional_input* = 0.

NOTE 7 r zeroes. *additional_input_string* is added only on the first iteration.

13. $i = i + 1$.

14. *reseed_counter* = *reseed_counter* + 1.

15. If (|*temp*| < *requested_no_of_bits*), then go to step 7.

16. *pseudorandom_bits* = **Truncate** (*temp*, $i \times k$, *requested_no_of_bits*).

17. Update the changed values in the *state*.

    17.1 *state*(*state_handle*).S = S.

    17.2 *state*(*state_handle*).*reseed_counter* = *reseed_counter*.

18. **Return** ("Success", *pseudorandom_bits*).

C.4.3.2.6 Inserting additional entropy into the state of MS_DRBG (…)

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

1. calling the **Reseed_MS_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. utilising the automatic reseeding feature of the **MS_DRBG (…).** The automatic reseeding feature forces a call to **Reseed_MS_DRBG_Instantiation (…)** whenever 50000 blocks of output have been produced since the last reseeding;

3. setting *prediction_resistance_flag* = 1 at instantiation. This forces a call to **Reseed_MS_DRBG_Instantiation (…)** each time **MS_DRBG (…)** is invoked; or

4. supplying additional input during any call to **MS_DRBG (…)** for random bits.

NOTE Frequent calls to the **Get_entropy (…)** function may cause severe performance degradation with this or any DRBG.

C.5 DRBG based on multivariate quadratic equations

C.5.1 Introduction to a DRBG based on multivariate quadratic equations

MQ_DRBG (…) relies on the iteration of a randomly chosen multivariate quadratic system. Instantiation and reseeding is based on the hash-based derivation function **Hash_df** to distribute entropy throughout the seed. The function **Hash_df** is specified in C.2.2.2.2 and makes use of an ISO/IEC approved hash function complying with the security strength required to instantiate **MQ_DRBG (…)**.

C.5.2 Multivariate Quadratic DRBG (MQ_DRBG)

C.5.2.1 Discussion

MQ_DRBG (…) is based on the following hard problem: Given a system P of multivariate quadratic equations over a binary field and the result $P(x)$ of the evaluation of the system on an input x, find x. (This problem is sometimes referred to as the *multivariate quadratic problem*).

MQ_DRBG (…) uses a seed of *state_length* bits to initiate the generation of *block_length*-bit pseudorandom strings by reiterating a multivariate quadratic system mapping *state_length*-bit strings to (*state_length* + *block_length*)-bit strings. The length parameters *state_length* and *block_length* are chosen as multiples of 8 close to each other. Selecting length parameters as small as possible, subject to the security strength required by the application, may result in improved performance.

The instantiation of this DRBG requires the selection of an appropriate system of multivariate quadratic equations as specified in C.5.2.5 for the desired security strength.

The *seed* used to define the initial value (S) of the DRBG shall have entropy that is at least the maximum of 128 and the desired security *strength* (i.e., entropy ≥ **max** (128, strength)). The seed length shall be *state_length* bits. Backward secrecy is inherent in the algorithm, even if the internal state is compromised. Forward secrecy is also inherent when observed from outside the DRBG boundary. If an application is concerned about compromise of the hidden state in an instantiation of the **MQ_DRBG (…)**, the state may be infused with new entropy in a number of ways, as discussed in C.5.2.2.7.

When optional additional input (*additional_input*) is used, the value of *additional_input* is hashed to a *state_length*-bit string.

C.5.2.2 Description

C.5.2.2.1 General

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

1. a value S that is updated during each call to the DRBG;
2. the length of that value *state_length*;
3. a block length parameter *block_length*;
4. a parameter *field_size*;
5. a system parameter P providing the coefficients of a system of multivariate quadratic equations;
6. a counter *reseed_counter* that indicates the number of blocks of pseudorandom bits generated since the last entropy input was added to the state during instantiation or reseeding;
7. a maximum value *reseed_interval* for *reseed_counter*;
8. the security *strength* of the DRBG instantiation; and
9. a value *prediction_resistance_flag* that indicates whether or not the forward secrecy capability is required by the DRBG.

The variables used in the description of **MQ_DRBG (…)** are:

additional_input	Optional additional input.
block	A temporary bitstring.
block_length	The number of bits returned by the **Evaluate_MQ(…)** function that serve as pseudorandom bits returned by **MQ_DRBG(…)**. This parameter will always be at least *requested_block_length*, and will always be at least *strength*.
entropy_input	The bits containing entropy that are used to determine *seed_material* and to generate a seed.
field_size	The number of bits of an element of the Galois field GF(2^{field_size}).
field_vector(*bits*, *in_len*, *field_size***)**	A function that inputs the bitstring *bits* of length *in_len* that is a multiple of *field_size*. It returns an array *vec* of length *in_len* / *field_size* whose elements are bitstrings of length *field_size*, where *vec*[1] = *bits*[1] || … || *bits*[*field_size*], *vec*[2] = *bits*[*field_size*+1] || … || *bits*[2**field_size*], and so on, so that **flatten(…)** applied to the output returns the original bitstring.

flatten(*in_array*, *num_elements***)**	A function that inputs an array of *num_elements* bitstrings and returns the bitstring *in_array*[1] || *in_array*[2] ||…|| *in_array*[*num_elements*] made of their concatenation in their order of enumeration.
Get_entropy (*min_entropy*, *min_length*, *max_length***)**	A function that acquires a string of bits from an entropy 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 (*hash_input***)**	An ISO/IEC approved hash-function that shall be as specified in ISO/IEC 10118-3, returning a bitstring whose input *hash_input* may be any multiple of 8 bits in length.
Hash_df (*hash_input*, *output_len***)**	A function to distribute the entropy in *hash_input* to a bitstring *output_len* long. The function **Hash (…)** is used to do this. *hash_input* may be any multiple of 8 bits in length; *output_len* is arbitrary.
i, j, k	Temporary values used as loop counters.
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.
min_entropy	A value used in the request to **Get_entropy(…)** to indicate the minimum entropy to be provided for seeding material. This value will always be fixed to **max**(128, *strength*).
min_length	The minimum length of the *entropy_input*.
n, m	Temporary integer values used as loop bounds.
Null	The null (i.e., empty) string.
P	A bitstring used to store the coefficients of the system of multivariate quadratic equations used by the DRBG.
P_vec	Vector of field elements used as a temporary variable.
pad8(*bitstring***)**	A function that inputs an arbitrary length bitstring and returns a copy of that bitstring padded on the right with binary 0's, if necessary, to a multiple of 8.
personalisation_string	A byte array that can provide additional assurance of seed uniqueness at instantiation.
prediction_resistance_flag	An instantiation flag indicating whether or not forward secrecy is to be provided by the DRBG. By default, this flag is not set. Setting this flag to 1 forces a call to **Reseed_MQ_DRBG_Instantiation (…)** from within **MQ_DRBG (…)** each time the process is invoked.
pseudorandom_bits	The pseudorandom bits produced by the DRBG.
requested_block_length	The requested value for *block_length* at initialisation.
requested_no_of_bits	The number of pseudorandom bits to be returned on a call to **MQ_DRBG (…)**.

requested_strength	The security strength to be associated with the requested pseudorandom bits.
reseed_counter	A count of the number of iterations of the **MQ_DRBG (…)** since the last reseeding.
reseed_interval	The maximum number of blocks of random output produced before the DRBG shall be reseeded.
S	A value that is initialised at instantiation and reseeding and is updated by the **Evaluate_MQ(…)** function.
seed_material	The seed used to derive the initial value of S.
state (*state_handle*)	An array of states for different DRBG instantiations. A *state* is carried between DRBG calls. For the **MQ_DRBG (…)**, the *state* for an instantiation is defined as *state* (*state_handle*) = { S, P, *reseed_counter*, *reseed_interval*, *state_length*, *block_length*, *field_size*, *strength*, *prediction_resistance_flag* }. A particular element of the *state* is specified as *state*(*state_handle*).*element*, e.g., *state* (*state_handle*).S.
state_handle	A handle to the state space for the given instantiation.
state_length	The size of the state value S within the DRBG. This value will always be at least *requested_block_length*, and will always be at least *strength*.
status	The *status* returned from a function call, where *status* = “Success” or a failure message.
strength	The security strength provided by the DRBG instantiation.
system_length	The size of the bitstring P that contains the system of quadratic equations used by **MQ_DRBG (…)**. It is completely determined by the selection of the parameters *state_length*, *block_length* and *field_size* as discussed in C.5.2.4.
t	Temporary integer value used as an index into bitstrings.
temp	A temporary bitstring.
Truncate(*bits*, *in_len*, *out_len*)	A function that inputs a bitstring of *in_len* bits, returning a string consisting of the leftmost *out_len* bits of input. If *in_len* < *out_len*, the input string is padded on the right with (*out_len* - *in_len*) zeroes, and the result is returned.
x_vec, *y_vec*, *z_vec*	Vectors of field elements used as temporary variables.
$x[\,i\,]$	For an array x of bitstrings, the i-th element of the array.
*	Multiplication over the binary field of size *field_size*.

C.5.2.2.2 Selection of the MQ_DRBG parameters

The instantiation of **MQ_DRBG (…)** requires the appropriate selection of instantiation parameters. Selecting appropriate parameters is performed based on the values of *requested_strength* and *requested_block_length* given as inputs to the instantiation function. The parameters are selected according to the following table.

Table C.5 — MQ_DRBG instantiation parameters based on *requested_block_length* and *requested_strength*

requested_strength	*requested_block_length*			
	1-112	113-128	129-192	193-256
1-80	*strength*=80 *state_length*=112 *block_length*=112 *field_size*=1 *reseed_interval*=2^{23} *system_length*=1417696	*strength*=80 *state_length*=128 *block_length*=128 *field_size*=4 *reseed_interval*=2^{12} *system_length*=143616	*strength*=80 *state_length*=192 *block_length*=192 *field_size*=6 *reseed_interval*=2^{12} *system_length*=215424	*strength*=80 *state_length*=256 *block_length*=256 *field_size*=8 *reseed_interval*=2^{14} *system_length*=287232
81-112	*strength*=112 *state_length*=120 *block_length*=112 *field_size*=1 *reseed_interval*=2^{26} *system_length*=1684552	*strength*=112 *state_length*=128 *block_length*=128 *field_size*=1 *reseed_interval*=2^{32} *system_length*=2113792	*strength*=112 *state_length*=192 *block_length*=192 *field_size*=4 *reseed_interval*=2^{12} *system_length*=470400	*strength*=112 *state_length*=256 *block_length*=256 *field_size*=4 *reseed_interval*=2^{21} *system_length*=1098240
113-128	ERROR	*strength*=128 *state_length*=128 *block_length*=128 *field_size*=1 *reseed_interval*=2^{28} *system_length*=2113792	*strength*=128 *state_length*=192 *block_length*=192 *field_size*=3 *reseed_interval*=2^{16} *system_length*=823680	*strength*=128 *state_length*=256 *block_length*=256 *field_size*=4 *reseed_interval*=2^{17} *system_length*=1098240
129-192	ERROR	ERROR	*strength*=192 *state_length*=200 *block_length*=192 *field_size*=1 *reseed_interval*=2^{32} *system_length*=7879592	*strength*=192 *state_length*=256 *block_length*=256 *field_size*=2 *reseed_interval*=2^{30} *system_length*=4293120
193-256	ERROR	ERROR	ERROR	*strength*=256 *state_length*=272 *block_length*=256 *field_size*=1 *reseed_interval*=2^{32} *system_length*=19604112

C.5.2.2.3 Instantiation of MQ_DRBG (…)

The following process or its equivalent shall be used to initialise the **MQ_DRBG (…)** process.

Instantiate_MQ_DRBG (…):

Input: integer (*requested_strength*, *prediction_resistance_flag*, *requested_block_length*), string *personalisation_string*.

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

Process:

1. If *requested_strength* or *requested_block_length* is larger than the maximum that can be provided by the implementation, then **Return**(Failure message). If *requested_block_length* is less than *requested_strength* then **Return**(Failure message).

2. Determine parameters of the state space. Round *requested_strength* up to the next available size on Table C.5. Set *strength*, *state_length*, *block_length*, *field_size* and *reseed_interval* to the values found on this table.

3. Select the system parameter P according to *state_length* and *block_length*.

NOTE The format and selection of P is described in C.5.2.4 and C.5.2.5 respectively.

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

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

6. If (*status* $\neq$ "Success"), then **Return** (Failure message).

7. *seed_material* = *entropy_input* || *personalisation_string*.

8. S = **Hash_df** (**pad8**(*seed_material*), *state_length*).

9. Initialise *reseed_counter* to 0.

10. *state*(*state_handle*) = { S, P, *reseed_counter*, *reseed_interval*, *state_length*, *block_length*, *field_size*, *strength*, *prediction_resistance_flag* }.

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

C.5.2.2.4 Reseeding MQ_DRBG (…) Instantiation

The following process or its equivalent shall be used to reseed the **MQ_DRBG (…)** process, after it has been instantiated.

Reseed_MQ_DRBG_Instantiation (…):

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

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

Process:

1. If a *state* is not available, **Return** (Failure message).

2. Get the appropriate state values.

   2.1 *strength* = *state*(*state_handle*).*strength*.

   2.2 *state_length* = *state*(*state_handle*).*state_length*.

   2.3 S = *state*(*state_handle*).S.

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

4. (*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *state_length*, *max_length*).

5. If (*status* ≠ "Success"), then **Return** (Failure message).
6. *seed_material* = S || *entropy_input* || *additional_input*.
7. S = **Hash_df** (**pad8**(*seed_material*), *state_length*).
8. Update the appropriate state values.
   - 8.1 *state*(*state_handle*).S = S.
   - 8.2 *state*(*state_handle*).*reseed_counter* = 0.
9. **Return** ("Success").

C.5.2.2.5 Generating pseudorandom bits using MQ_DRBG (…)

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

MQ_DRBG (…):

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

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

Process:

1. If a *state* is not available, **Return** (Failure message, *Null*).
2. Get the appropriate *state* values.
   - 2.1 *strength* = *state*(*state_handle*).*strength*.
   - 2.2 *state_length* = *state*(*state_handle*).*state_length*.
   - 2.3 *block_length* = *state*(*state_handle*).*block_length*.
   - 2.4 *field_size* = *state*(*state_handle*).*field_size*.
   - 2.5 S = *state*(*state_handle*).S.
   - 2.6 P = *state*(*state_handle*).P.
   - 2.7 *reseed_counter* = *state*(*state_handle*).*reseed_counter*.
   - 2.8 *reseed_interval* = *state*(*state_handle*).*reseed_interval*.
   - 2.9 *prediction_resistance_flag* = *state*(*state_handle*).*prediction_resistance_flag*.
3. If (*requested_strength* > *strength*), then **Return** (Failure message, *Null*).
4. *temp* = the *Null* string; i = 0.
5. If (*prediction_resistance_flag* = 1), then:
   - 5.1 *status* = **Reseed_MQ_DRBG_Instantiation** (*state_handle*, *additional_input*).
   - 5.2 If (*status* ≠ "Success"), then **Return** (*status*, *Null*).

5.3 *additional_input* = *Null*.

6. If (*reseed_counter* $\geq$ *reseed_interval*), then:

   6.1 *status* = **Reseed_MQ_DRBG_Instantiation** (*state_handle*, *additional_input*).

   6.2 If (*status* $\neq$ "Success"), then **Return** (*status*, *Null*).

   6.3 *additional_input* = *Null*.

7. If (*additional_input* $\neq$ *Null*), then:

   7.1 *additional_input* = **Hash_df** (**pad8**(*additional_input*), *state_length*).

   7.2 $S = S \oplus$ *additional_input*.

   7.3 *additional_input* = *Null*.

8. (S, *block*) = **Evaluate_MQ** (*state_length*, *block_length*, *field_size*, P, S).

9. *temp* = *temp* || *block*.

10. $i = i + 1$.

11. *reseed_counter* = *reseed_counter* + 1.

12. If (|*temp*| < *requested_no_of_bits*), then go to step 5.

13. *pseudorandom_bits* = **Truncate** (*temp*, $i \times$ *block_length*, *requested_no_of_bits*).

14. Update changed values in the *state*.

   14.1 *state*(*state_handle*).$S = S$.

   14.2 *state*(*state_handle*).*reseed_counter* = *reseed_counter*.

15. **Return** ("Success", *pseudorandom_bits*).

C.5.2.2.6 Evaluating multivariate quadratic equations using Evaluate_MQ (…)

The following process or its equivalent shall be used to evaluate a system of quadratic equations.

Evaluate_MQ (…):

Input: integer (*state_length*, *block_length*, *field_size*), bitstring (P, x).

Output: bitstring (y, z).

Process:

1. Convert bitstrings to vectors of field elements.

   1.1 *P_vec* = **field_vector**(P, *system_length*, *field_size*).

   1.2 *x_vec* = **field_vector**(x, *state_length*, *field_size*).

2. Initialise temporary vectors *y_vec* and *z_vec*.

2.1 *y_vec* = **field_vector**(0^{state_length}, *state_length*, *field_size*).

2.2 *z_vec* = **field_vector**(0^{block_length}, *block_length*, *field_size*).

3. n = *state_length* / *field_size*.

4. m = *block_length* / *field_size*.

5. t = 1.

6. For i = 1 to n do

6.1 For j = 1 to n do

6.1.1 For k = j to n do

6.1.1.1 *y_vec*[i] = *y_vec*[i] ⊕ (*P_vec*[t] ∗ *x_vec*[j] ∗ *x_vec*[k]).

6.1.1.2 t = t + 1.

6.2 If (*field_size* > 1), then:

6.2.1 For j = 1 to n do

6.2.1.1 *y_vec*[i] = *y_vec*[i] ⊕ (*P_vec*[t] ∗ *x_vec*[j]).

6.2.1.2 t = t + 1.

6.3 *y_vec*[i] = *y_vec*[i] ⊕ *P_vec*[t].

6.4 t = t + 1.

7. For i = 1 to m do

7.1 For j = 1 to n do

7.1.1 For k = j to n do

7.1.1.1 *z_vec*[i] = *z_vec*[i] ⊕ (*P_vec*[t] ∗ *x_vec*[j] ∗ *x_vec*[k]).

7.1.1.1 t = t + 1.

7.2 If (*field_size* > 1), then:

7.2.1 For j = 1 to n do

7.2.1.1 *z_vec*[i] = *z_vec*[i] ⊕ (*P_vec*[t] ∗ *x_vec*[j]).

7.2.1.2 t = t + 1.

7.3 *z_vec*[i] = *z_vec*[i] ⊕ *P_vec*[t].

7.4 t = t + 1.

8. Convert vectors to bitstrings.

8.1 y = **flatten**(*y_vec*, n).

8.2. z = **flatten**(*z_vec*, m).

9. **Return** (y, z).

NOTE 1 The input bitstring x contains *state_length* bits; the output bitstrings y, z contain *state_length* and *block_length* bits respectively. The coefficients of the quadratic equations that produce y and z are stored in the bitstring P. See C.5.2.3 and C.5.2.4 for a description of the format of P.

NOTE 2 The multiplication operation $*$ refers to the product in the binary field GF(2^{field_size}). Elements of the binary field are represented and handled as *field_size*-bit strings using the specific irreducible polynomials that are found in Table C.6. See C.5.2.3 for more detail on the representation of field elements as bitstrings. Additions in GF(2^{field_size}) are performed using the exclusive-or operation between two bitstrings of length *field_size*.

C.5.2.2.7 Inserting additional entropy into the state of MQ_DRBG (…)

Additional entropy may be inserted into the state of the **MQ_DRBG (…)** in three ways. Additional entropy may be inserted by:

1. calling the **Reseed_MQ_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. utilising the automatic reseeding feature of the **MQ_DRBG (…)**. The automatic reseeding feature will force a call to **Reseed_MQ_DRBG_Instantiation (…)** for new entropy whenever *reseed_interval* blocks of random have been output since the last call to reseed;

3. setting *prediction_resistance_flag* = 1 at instantiation. This forces a call to **Reseed_MQ_DRBG_Instantiation (…)** each time that **MQ_DRBG (…)** is invoked.

NOTE Frequent calls to the **Get_entropy (…)** function may cause severe performance degradation with this or any DRBG.

C.5.2.3 Format for representing field elements

An element of GF(2^{field_size}) is a univariate polynomial over GF(2) modulo the irreducible polynomial found in Table C.6. Therefore a field element is uniquely identified as the list of its coefficients (which are bits) or equivalently as a *field_size*-bit string composed of its GF(2) coefficients ordered by decreasing degree. This string representation is adopted for the storage and computations on field elements in **MQ_DRBG(…)**.

NOTE 1 As an example, $x^3 + x + 1$ in the field GF(2^4) is represented as the bitstring 1011.

Since field elements are stored and handled as bitstrings, every operation taking place on the binary field GF(2^{field_size}) is viewed as an operation on bitstrings. Hence field multiplication and addition on GF(2^{field_size}) take as input two *field_size*-bit strings and return a *field_size*-bit string.

NOTE 2 A field element is a single bit when *field_size* = 1.

Table C.6 — Irreducible polynomial for the binary field GF(2^{field_size})

field_size	Irreducible polynomial
1	$x + 1$
2	$x^2 + x + 1$
3	$x^3 + x + 1$
4	$x^4 + x + 1$
6	$x^6 + x + 1$
8	$x^8 + x^4 + x^3 + x^2 + 1$

C.5.2.4 Format for representing systems of multivariate quadratic equations

The bitstring P included in the state space contains the coefficients of the system of multivariate quadratic equations used within **MQ_DRBG (…)**. Each coefficient is an element of the binary field GF(2^{field_size}) as is therefore handled as a *field_size*-bit string. All coefficients are concatenated as described below to form the bitstring P. Hence the size of P is either

$$system\_length = (n + m) * (n * (n + 3) / 2 + 1) * field\_size$$

if *field_size* > 1 or

$$system\_length = (n + m) * (n * (n + 1) / 2 + 1) * field\_size$$

for *field_size* = 1, where $n = state\_length / field\_size$ and $m = block\_length / field\_size$ are respectively the number of field elements contained in the DRBG state and an output block generated by **Evaluate_MQ (…)**.

The coefficients of a quadratic equation are concatenated in lexicographic order and by decreasing degree. Namely, the coefficient of the term x_1x_1 appears first, followed by that of the term x_1x_2, and so forth up to the coefficient of the term x_1x_n. The coefficient of the term x_2x_2 appears next, followed by the one of the term x_2x_3, and so on, until the last quadratic coefficient is reached (the one of the term $x_{n-1}x_n$). Then linear coefficients appear, starting with the coefficient of the term x_1 and ending with the one of x_n. When *field_size* = 1, these coefficients are avoided since they are redundant with the quadratic coefficients of terms of the form $x_ix_i = x_i$. The string ends with the constant coefficient of the quadratic equation.

The coefficients of the system of quadratic equations are concatenated in sequential order, starting with the coefficients of the first equation and ending with the ones of the $(n + m)$-th equation. The resulting *system_length*-bit string forms the system parameter P.

C.5.2.5 Selection of appropriate systems of multivariate quadratic equations

Weak instances of systems of multivariate quadratic equations are known to exist. The rank of a quadratic equation is the rank of the matrix whose entry a_{ij} is the coefficient of the term x_ix_j in the quadratic equation, for $i \neq j$, and 0 for $i = j$. The output of a low-rank quadratic equation is biased toward 0. Similarly, if a linear combination of quadratic equations has low rank, then the outputs of those quadratic equations are correlated in the sense that a linear combination of their output is biased toward zero.

A randomly selected system of quadratic equations used in an instance of **MQ_DRBG (…)** is very likely to have the property that no quadratic equation or low-weight linear combination of quadratic equations has low rank. This shall be accomplished by obtaining the values *min_rank* and *max_weight* from Table C.7, and verifying the two following criteria:

1. All multivariate quadratic equations have rank at least *min_rank*, and

2. All sums of at most *max_weight* multivariate quadratic equations have rank at least *min_rank*.

Table C.7 — System selection parameters based on *requested_strength* and *requested_block_length*

requested_strength	*requested_block_length*			
	1-112	113-128	129-192	193-256
1 - 80	*min_rank* = 106 *max_weight* = 4	*min_rank* = 30 *max_weight* = 5	*min_rank* = 30 *max_weight* = 5	*min_rank* = 30 *max_weight* = 5
81 - 112	*min_rank* = 114 *max_weight* = 4	*min_rank* = 122 *max_weight* = 4	*min_rank* = 44 *max_weight* = 5	*min_rank* = 60 *max_weight* = 5
113 - 128	ERROR	*min_rank* = 122 *max_weight* = 4	*min_rank* = 60 *max_weight* = 5	*min_rank* = 60 *max_weight* = 5
129 - 192	ERROR	ERROR	*min_rank* = 192 *max_weight* = 4	*min_rank* = 124 *max_weight* = 4
193 - 256	ERROR	ERROR	ERROR	*min_rank* = 264 *max_weight* = 4

Annex D
(normative)

Application specific constants

D.1 Constants for the Dual_EC_DRBG

D.1.1 Introduction to Dual_EC_DRBG required constants

The **Dual_EC_DRBG (…)** requires the specifications of an elliptic curve and two points on the elliptic curve. One of the following ISO/IEC approved curves and points shall be used in applications requiring certification under ISO/IEC 19790.

Curves given in D.1.2 are mod p curves.

Curves given in D.1.3 are binary curves.

D.1.2 Curves over prime fields

D.1.2.1 Introduction to curves over prime fields

Each of following mod *p* curves is given by the equation:

$$y^2 = x^3 - 3x + b \pmod{p}$$

Notation:

p - order of the field F_p, given in decimal

r - order of the Elliptic Curve Group, in decimal . Note that r is used here, but is referred to as n in the description of the **Dual_EC_DRBG( )**

b - coefficient above

The x and y coordinates of the base point, i.e., generator G, are the same as for the point P.

D.1.2.2 Curve P-192

p = 6277101735386680763835789423207666416083908700390324961279

r = 6277101735386680763835789423176059013767194773182842284081

b = 64210519 e59c80e7 0fa7e9ab 72243049 feb8deec c146b9b1

Px = 188da80e b03090f6 7cbf20eb 43a18800 f4ff0afd 82ff1012

Py = 07192b95 ffc8da78 631011ed 6b24cdd5 73f977a1 1e794811

Qx = c7f6ebec 5174ea4e 813ac769 31aedec3 b87d6f67 792f358e

Qy = 8b0bdab7 02e9c962 68e1cef5 f34033b9 71bc41a5 d669d8e4

D.1.2.3 Curve P-224

p = 26959946667150639794667015087019630673557916
260026308143510066298881

r = 26959946667150639794667015087019625940457807
714424391721682722368061

b = b4050a85 0c04b3ab f5413256 5044b0b7 d7bfd8ba 270b3943 2355ffb4

Px = b70e0cbd 6bb4bf7f 321390b9 4a03c1d3 56c21122 343280d6 115c1d21

Py = bd376388 b5f723fb 4c22dfe6 cd4375a0 5a074764 44d58199 85007e34

Qx = 68623591 6e11adfa f080a451 477fa27a f21248be 916d3458 a583a3c9

Qy = 6060018a 24b35be6 caecf3f0 7f2c6b43 4e47479e 55362c8f 5707adca

D.1.2.4 Curve P-256

p = 115792089210356248762697446949407573530086143415290314195533631308867097853951

r = 115792089210356248762697446949407573529996955224135760342422259061068512044369

b = 5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f6 3bce3c3e 27d2604b

Px = 6b17d1f2 e12c4247 f8bce6e5 63a440f2 77037d81 2deb33a0 f4a13945 d898c296

Py = 4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16 2bce3357 6b315ece cbb64068 37bf51f5

Qx = c97445f4 5cdef9f0 d3e05e1e 585fc297 235b82b5 be8ff3ef ca67c598 52018192

Qy = b28ef557 ba31dfcb dd21ac46 e2a91e3c 304f44cb 87058ada 2cb81515 1e610046

D.1.2.5 Curve P-384

p = 39402006196394479212279040100143613805079739
27046544666794829340424572177149687032904726
60882589380018616069731123 19

r = 39402006196394479212279040100143613805079739
27046544666794690527962765939911326356939895
63081522949135544336539426 43

b = b3312fa7 e23ee7e4 988e056b e3f82d19 181d9c6e fe814112 0314088f
5013875a c656398d 8a2ed19d 2a85c8ed d3ec2aef

Px = aa87ca22 be8b0537 8eb1c71e f320ad74 6e1d3b62 8ba79b98 59f741e0
82542a38 5502f25d bf55296c 3a545e38 72760ab7

Py = 3617de4a 96262c6f 5d9e98bf 9292dc29 f8f41dbd 289a147c e9da3113
b5f0b8c0 0a60b1ce 1d7e819d 7a431d7c 90ea0e5f

Qx = 8e722de3 125bddb0 5580164b fe20b8b4 32216a62 926c5750 2ceede31
c47816ed d1e89769 124179d0 b6951064 28815065

Qy = 023b1660 dd701d08 39fd45ee c36f9ee7 b32e13b3 15dc0261 0aa1b636
e346df67 1f790f84 c5e09b05 674dbb7e 45c803dd

D.1.2.6 Curve P-521

p = 68647976601306097149819007990813932172694353
00143305409394463459185543183397656052122559
64066145455497729631139148085803712198799971
66438125740282911150571 51

r = 68647976601306097149819007990813932172694353
00143305409394463459185543183397655394245057
74633321719753296399637136332111386476861244
03803403728088927070054 49

b = 051953eb 9618e1c9 a1f929a2 1a0b6854 0eea2da7 25b99b31 5f3b8b48
9918ef10 9e156193 951ec7e9 37b1652c 0bd3bb1b f073573d f883d2c3
4f1ef451 fd46b503 f00

Px = c6858e06 b70404e9 cd9e3ecb 662395b4 429c6481 39053fb5 21f828af
606b4d3d baa14b5e 77efe759 28fe1dc1 27a2ffa8 de3348b3 c1856a42
9bf97e7e 31c2e5bd 66

Py = 11839296 a789a3bc 0045c8a5 fb42c7d1 bd998f54 449579b4 46817afb
d17273e6 62c97ee7 2995ef42 640c550b 9013fad0 761353c7 086a272c
24088be9 4769fd16 650

Qx = 1b9fa3e5 18d683c6 b6576369 4ac8efba ec6fab44 f2276171 a4272650
7dd08add 4c3b3f4c 1ebc5b12 22ddba07 7f722943 b24c3edf a0f85fe2
4d0c8c01 591f0be6 f63

Qy = 1f3bdba5 85295d9a 1110d1df 1f9430ef 8442c501 8976ff34 37ef91b8
1dc0b813 2c8d5c39 c32d0e00 4a3092b7 d327c0e7 a4d26d2c 7b69b58f
90666529 11e45777 9de

D.1.3 Curves over binary fields

D.1.3.1 Introduction to curves over binary fields

For each field degree m, a pseudo random curve (B) is given, along with a Koblitz curve (K)

The pseudo random curve has the form

$$E: y^2 + xy = x^3 + x^2 + b,$$

and the Koblitz curve has the form:

$$E: y^2 + xy = x^3 + ax^2 + 1, \text{ where } a = 0 \text{ or } 1.$$

For each pseudorandom curve, the cofactor is $f = 2$. The cofactor of each Koblitz curve is $f = 2$ if $a = 1$ and $f = 4$ if $a = 0$.

The coefficients of the pseudo random curves, and the coordinates of the points P and Q for both kinds of curves, are given in terms of both the polynomial and normal basis representations in hex.

For each *m*, the following parameters are given for the Field Representation – the normal basis type *T* and the field polynomial *p*(*t*) (a trinomial or pentanomial).

The order *r* of the base point *P* is given in decimal. Note that *r* is used here whereas in FIPS 186-3, it is referred to as *n* in the description of the **Dual_EC_DRBG ( )**.

NOTE Additional details may found in FIPS 186-3 (DSS). [14]

D.1.3.2 Degree 163 binary field

D.1.3.2.1 The degree 163 binary field representation parameters

$T = 4$

$p(t) = t^{163} + t^7 + t^6 + t^3 + 1$

D.1.3.2.2 Curve K-163

$a = 1$

$r = 5846006549323611672814741753598448348329118574063$

Polynomial Basis:

Px = 00000002 fe13c053 7bbc11ac aa07d793 de4e6d5e 5c94eee8

Py = 00000002 89070fb0 5d38ff58 321f2e80 0536d538 ccdaa3d9

Normal Basis:

Px = 00000000 5679b353 caa46825 fea2d371 3ba450da 0c2a4541

Py = 00000002 35b7c671 00506899 06bac3d9 dec76a83 5591edb2

Polynomial Basis:

Qx = 00000003 3fb85544 30126994 35eb2808 89e90c23 4a0d4676

Qy = 00000003 a6a3de4f 08b62c18 f74c2b97 4cb70d58 ddc92854

Normal Basis:

Qx = 00000001 5653b6cd f9f578d2 6462cd09 ce97eb04 97b4f5a8

Qy = 00000002 dd9bd677 b4bde91b a928f351 09c34341 d845eefc

D.1.3.2.3 Curve B-163

r = 5846006549323611672814742442876390689256843201587

Polynomial Basis:

b = 2 0a601907 b8c953ca 1481eb10 512f7874 4a3205fd

Px = 3 f0eba162 86a2d57e a0991168 d4994637 e8343e36

Py = 0 d51fbc6c 71a0094f a2cdd545 b11c5c0c 797324f1

Normal Basis:

b = 6 645f3cac f1638e13 9c6cd13e f61734fb c9e3d9fb

Px = 0 311103c1 7167564a ce77ccb0 9c681f88 6ba54ee8

Py = 3 33ac13c6 447f2e67 613bf700 9daf98c8 7bb50c7f

Polynomial Basis:

Qx = 5 7f4128d8 02fd4e34 2bd8ecd2 d7da7337 84502352

Qy = 2 69575de3 c75e9c90 d30fb407 857e5383 581b83de

Normal Basis:

Qx = 7 52425a4c 34d5db83 7573e705 35d33eed 299719eb

Qy = 5 b9e83817 fa6409eb de7c7bbb 880ba9a8 63cfd1d5

D.1.3.3 Degree 233 binary field

D.1.3.3.1 The degree 233 binary field representation parameters

$T = 2$

$p(t) = t^{233} + t^{74} + 1$

D.1.3.3.2 Curve K-233

a = 0

r = 3450873173395281893717377931138512760570940988862252126328087024741343

Polynomial Basis:

Px = 00000172 32ba853a 7e731af1 29f22ff4 149563a4 19c26bf5 0a4c9d6e efad6126

Py = 000001db 537dece8 19b7f70f 555a67c4 27a8cd9b f18aeb9b 56e0c110 56fae6a3

Normal Basis:

Px= 000000fd e76d9dcd 26e643ac 26f1aa90 1aa12978 4b71fc07 22b2d056 14d650b3

Py = 00000064 3e317633 155c9e04 47ba8020 a3c43177 450ee036 d6335014 34cac978

Polynomial Basis:

Qx = 000000aa 7178e973 8a6f797a 1c265465 06106896 0a58b3fe a3afc77f 18404eee

Qy = 0000002d 12a8f3e9 884bf31d 052a8eaf 414b891a 0a40491e 1f9d2576 79248ee2

Normal Basis:

Qx = 0000015a 96493d91 e56b5f10 579a7d58 eb895e06 8d94e1af 86d34143 4377548c

Qy = 0000006b 13a689bb 3730dfd7 a46486ea ff8eb6cb 9d815981 a927d2eb 8cfa9b00

D.1.3.3.3 Curve B-233

r = 690174634679056378743475586227702555583981273734501355379383634485463

Polynomial Basis:

b = 066 647ede6c 332c7f8c 0923bb58 213b333b 20e9ce42 81fe115f 7d8f90ad

Px = 000000fa c9dfcbac 8313bb21 39f1bb75 5fef65bc 391f8b36 f8f8eb73 71fd558b

Py = 00000100 6a08a419 03350678 e58528be bf8a0bef f867a7ca 36716f7e 01f81052

Normal Basis:

b = 1a0 03e0962d 4f9a8e40 7c904a95 38163adb 82521260 0c7752ad 52233279

Px = 0000018b 863524b3 cdfefb94 f2784e0b 116faac5 4404bc91 62a363ba b84a14c5

Py = 00000049 25df77bd 8b8ff1a5 ff519417 822bfedf 2bbd7526 44292c98 c7af6e02

Polynomial Basis:

Qx = 000000cb 50ce04af f4ea6111 aaccfe04 ae5f0dfe 95a59db4 cd4aba0c 1126615a

Qy = 0000005b ab8a93a0 5c42caae 1b322b14 876ec2e0 5c994a25 8e67295e 5808eaf9

Normal Basis:

Qx = 00000055 ea07c1ca 4a4312f3 4562737c 257f4fa8 3b9d3d48 8a123cab 238f69a2

Qy = 00000055 d60ea17a 1cb969a8 3786a82f 8172e889 026195f9 923ba4b1 beeb5702