



# **American National Standard for Financial Services**

## **ANS X9.82: Part 3–2006**

### **Random Number Generation Part 3: Deterministic Random Bit Generators**



Accredited Standards Committee X9, Incorporated  
Financial Industry Standards

**Date Approved:**

**American National Standards Institute**

American National Standards, Technical Reports and Guides developed through Accredited Standards Committee X9, Inc. are copyrighted. Copying these documents for personal or commercial use outside X9 membership agreements is prohibited without express written permission of the Accredited Standards Committee X9, Inc. For additional information, please contact ASC X9, Inc., 1212 West Street, Suite 200, Annapolis, Maryland 21401

## Contents

Foreword .....	VII
Introduction .....	IX
1 Scope .....	1
2 Conformance.....	1
3 Normative References .....	2
4 Terms and Definitions .....	2
5 Abbreviations and Symbols.....	5
6 General Discussion and Organization .....	7
7 Functional Model .....	9
7.1 General Discussion .....	9
7.2 Functional Model Components .....	9
7.2.1 Entropy Input.....	9
7.2.2 Other Inputs.....	10
7.2.3 The Internal State.....	10
7.2.4 The DRBG Mechanism Functions .....	10
8. DRBG Mechanism Concepts and General Requirements .....	11
8.1 Introduction.....	11
8.2 DRBG Mechanism Functions and a DRBG Instantiation .....	11
8.2.1 DRBG Mechanism Functions .....	11
8.2.2 DRBG Instantiations .....	11
8.2.3 Internal States .....	11
8.2.4 Security Strengths Supported by an Instantiation .....	12
8.3 DRBG Mechanism Boundaries.....	13
8.4 Seeds .....	14
8.4.1 General Discussion .....	14
8.4.2 Generation and Handling of Seeds .....	14
8.5 Other Inputs to the DRBG Mechanism .....	17
8.5.1 Discussion.....	17
8.5.2 Personalization String.....	17

8.5.3	Additional Input.....	18
8.6	Prediction Resistance and Backtracking Resistance.....	18
9	<b>DRBG Mechanism Functions.....</b>	<b>19</b>
9.1	General Discussion .....	19
9.2	Instantiating a DRBG.....	19
9.3	Reseeding a DRBG Instantiation .....	22
9.4	Generating Pseudorandom Bits Using a DRBG.....	24
9.4.1	The Generate Function.....	24
9.4.2	Reseeding at the End of the Seedlife.....	27
9.4.3	Handling Prediction Resistance Requests .....	27
9.5	Removing a DRBG Instantiation .....	27
10	<b>DRBG Algorithm Specifications .....</b>	<b>29</b>
10.1	Overview.....	29
10.2	Deterministic RBG Based on Hash Functions .....	29
10.2.1	Discussion.....	29
10.2.2	HMAC_DRBG.....	30
10.2.2.1	Discussion .....	30
10.2.2.2	Specifications .....	31
10.2.2.2.1	HMAC_DRBG Internal State.....	31
10.2.2.2.2	The Update Function (CTR_DRBG_Update).....	31
10.2.2.2.3	Instantiation of HMAC_DRBG.....	32
10.2.2.2.4	Reseeding an HMAC_DRBG Instantiation.....	33
10.2.2.2.5	Generating Pseudorandom Bits Using HMAC_DRBG.....	33
10.3	<b>DRBG Mechanisms Based on Block Ciphers.....</b>	<b>35</b>
10.3.1	Discussion.....	35
10.3.2	CTR_DRBG .....	35
10.3.2.1	CTR_DRBG Description.....	35
10.3.2.2	Specifications .....	37
10.3.2.2.1	CTR_DRBG Internal State .....	37
10.3.2.2.2	The Update Function (CTR_DRBG_Update).....	38
10.3.2.2.3	Instantiation of CTR_DRBG .....	38
10.3.2.2.4	Reseeding a CTR_DRBG Instantiation.....	40

10.3.2.2.5	Generating Pseudorandom Bits Using CTR_DRBG .....	42
<b>10.4</b>	<b>DRBG Mechanisms Based on Number Theoretic Problems .....</b>	<b>46</b>
10.4.1	Discussion .....	46
10.4.2	Dual Elliptic Curve Deterministic RBG (Dual_EC_DRBG) .....	46
10.4.2.1	Discussion .....	46
10.4.2.2	Specifications .....	48
10.4.2.2.1	Dual_EC_DRBG Internal State .....	48
10.4.2.2.2	Instantiation of Dual_EC_DRBG .....	49
10.4.2.2.3	Reseeding of a Dual_EC_DRBG Instantiation .....	50
10.4.2.2.4	Generating Pseudorandom Bits Using Dual_EC_DRBG .....	50
<b>10.5</b>	<b>Auxilliary Functions .....</b>	<b>53</b>
10.5.1	Discussion .....	53
10.5.2	Derivation Function Using a Hash Function (Hash_df) .....	53
10.5.3	Derivation Function Using a Block Cipher Algorithm (Block_Cipher_df) .....	54
10.5.4	BCC Function .....	56
<b>11</b>	<b>Assurance .....</b>	<b>57</b>
11.1	Overview .....	57
11.2	Minimal Documentation Requirements .....	57
11.3	Implementation Validation Testing .....	58
11.4	Health Testing .....	58
11.4.1	Overview .....	58
11.4.2	Known-Answer Testing .....	59
11.4.3	Testing the Instantiate Function .....	59
11.4.4	Testing the Generate Function .....	59
11.4.5	Testing the Reseed Function .....	60
11.4.6	Testing the Uninstantiate Function .....	60
11.4.7	Error Handling .....	60
11.4.7.1	General Discussion .....	60
11.4.7.2	Errors Encountered During Normal Operation .....	60
11.4.7.3	Errors Encountered During Health Testing .....	61
<b>Annex A:</b>	<b>(Normative) Application-Specific Constants .....</b>	<b>62</b>

A.1	Constants for the Dual_EC_DRBG .....	62
A.1.1	Curves over Prime Fields.....	62
A.1.1.1	Curve P-256 .....	62
A.1.1.2	Curve P-384 .....	63
A.1.1.3	Curve P-521 .....	63
A.2	Using Alternative Points in the Dual_EC_DRBG().....	64
A.2.1	Generating Alternative $P, Q$ .....	64
A.2.2	Additional Self-testing Required for Alternative $P, Q$ .....	65
ANNEX B :	(Normative) Conversion and Auxilliary Routines .....	66
B.1	Bitstring to an Integer .....	66
B.2	Integer to a Bitstring .....	66
B.3	Integer to a Byte String .....	66
B.4	Byte String to an Integer .....	67
Annex C:	(Informative) Security Considerations .....	68
C.1	Extracting Bits in the Dual_EC_DRBG (...).....	68
C.1.1	Potential Bias Due to Modular Arithmetic for Curves Over $F_p$ .....	68
C.1.2	Adjusting for the Missing Bit(s) of Entropy in the $x$ Coordinates. ....	68
ANNEX D:	(Informative) DRBG Mechanism Selection .....	72
D.1	Choosing a DRBG Algorithm .....	72
D.2	HMAC_DRBG .....	72
D.3	CTR_DRBG.....	73
D.4	DRBGs Based on Hard Problems .....	74
D.5	Summary for DRBG Selection.....	75
ANNEX E:	(Informative) Example Pseudocode for Each DRBG Mechanism.....	76
E.1	Preliminaries .....	76
E.2	HMAC_DRBG Example .....	76
E.2.1	Discussion.....	76
E.2.2	Instantiation of HMAC_DRBG.....	77
E.2.3	Generating Pseudorandom Bits Using HMAC_DRBG .....	78
E.3	CTR_DRBG Example Using a Derivation Function.....	80

E.3.1	Discussion .....	80
E.3.2	The CTR_DRBG_Update Function .....	80
E.3.3	Instantiation of CTR_DRBG Using a Derivation Function .....	81
E.3.4	Reseeding a CTR_DRBG Instantiation Using a Derivation Function .....	83
E.3.5	Generating Pseudorandom Bits Using CTR_DRBG .....	84
E.4	CTR_DRBG Example Without a Derivation Function .....	86
E.4.1	Discussion .....	86
E.4.2	The CTR_DRBG_Update Function .....	87
E.4.3	Instantiation of CTR_DRBG Without a Derivation Function .....	87
E.4.4	Reseeding a CTR_DRBG Instantiation Without a Derivation Function .....	87
E.4.5	Generating Pseudorandom Bits Using CTR_DRBG .....	87
E.5	Dual_EC_DRBG Example .....	88
E.5.1	Discussion .....	88
E.5.2	Instantiation of Dual_EC_DRBG .....	89
E.5.3	Reseeding a Dual_EC_DRBG Instantiation .....	90
E.5.4	Generating Pseudorandom Bits Using Dual_EC_DRBG .....	91
<b>ANNEX F: (Informative) DRBG Provision of RBG Security Properties .....</b>		<b>94</b>
F.1	Introduction .....	94
F.2	Security Strengths .....	94
F.3	Entropy and Min-Entropy .....	94
F.4	Backtracking Resistance and Prediction Resistance .....	94
F.5	Indistinguishability and Unpredictability .....	94
F.6	Desired RBG Output Properties .....	94
F.7	Desired RBG Operational Properties .....	95
<b>ANNEX G: .....</b>		<b>97</b>
G.1	Overview .....	97
G.2	HMAC_DRBG .....	97
G.3	CTR_DRBG .....	97
G.4	Dual_EC_DRBG .....	98
<b>ANNEX H: (Informative) Bibliography .....</b>		<b>101</b>

## **Foreword**

The Accredited Standards Committee on Financial Services (ANSI X9) has developed several cryptographic standards to protect financial information. Many of these standards require the use of Random Number Generators to generate random and unpredictable cryptographic keys and other critical security parameters. This Standard, *Random Number Generation*, defines techniques for the generation of random numbers that are used when other ASC standards require the use of random numbers for cryptographic purposes.

While the techniques specified in this Standard are designed to generate random numbers, the Standard does not guarantee that a particular implementation is secure. It is the responsibility of the financial institution to put an overall process in place with the necessary controls to ensure that the process is securely implemented. Furthermore, the controls should include the application with appropriate validation tests in order to verify compliance with this Standard.

Approval of an American National Standard requires verification by ASC that the requirements for due process, consensus, and other criteria for approval have been met by the standards developer.

Consensus is established when, in the judgment of the ASC Board of Standards Review, substantial agreement has been reached by directly and materially affected interests. Substantial agreement means much more than a simple majority, but not necessarily unanimity. Consensus requires that all views and objections be considered, and that a concerted effort be made toward their resolution.

The use of American National Standards is completely voluntary; their existence does not in any respect preclude anyone, whether he has approved the standards or not from manufacturing, marketing, purchasing, or using products, processes, or procedures not conforming to the standards.

The American National Standards Institute does not develop standards and will in no circumstances give an interpretation of any American National Standard. Moreover, no person shall have the right or authority to issue an interpretation of an American National Standard in the name of the American National Standards Institute. Requests for interpretations should be addressed to the secretariat or sponsor whose name appears on the title page of this Standard.

**CAUTION NOTICE:** This American National Standard may be revised or withdrawn at any time. The procedures of the American National Standards Institute require that action be taken to reaffirm, revise, or withdraw this Standard no later than five years from the date of approval.

**Draft ANS X9.82, Part 3 - November 2006**

Published by

**Accredited Standards Committee X9 Incorporated**  
**Financial Industry standards**  
**P.O. Box 4035**  
**Annapolis, MD 21403 USA**  
**X9 Online <http://www.x9.org>**

Copyright © 2004 ASC X9, Inc.

All rights reserved.

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without prior written permission of the publisher. Published in the United States of America.



## **Introduction**

**NOTE** The user's attention is called to the possibility that compliance with this Standard may require use of an invention covered by patent rights.

By publication of this Standard, no position is taken with respect to the validity of this claim or of any patent rights in connection therewith. The patent holder has, however, filed a statement of willingness to grant a license under these rights on reasonable and nondiscriminatory terms and conditions to applicants desiring to obtain such a license. Details may be obtained from the standards developer.

Suggestions for the improvement or revision of this Standard are welcome. They should be sent to the X9 Committee Secretariat, Accredited Standards Committee X9, Inc., Financial Industry Standards, P.O. Box 4035, Annapolis, MD 21403 USA.

This Standard was processed and approved for submittal to ANSI by the Accredited Standards Committee on Financial Services, X9. Committee approval of the Standard does not necessarily imply that all the committee members voted for its approval.

The X9 committee had the following members:

[To be supplied], X9 Chairman  
Vincent DeSantis, X9 Vice-Chairman  
Cynthia Fuller, Executive Director  
Isabel Bailey, Managing Director

### **Organization Represented**

[X9 membership to be supplied]

### **Representative**

The X9F subcommittee on Data and Information Security had the following members:  
Richard J. Sweeney, Chairman

### **Organization Represented**

[X9F membership to be supplied]

### **Representative**

Under ASC X9 procedures, a working group may be established to address specific segments of work under the ASC X9 Committee or one of its subcommittees. A working group exists only to develop standard(s) or guideline(s) in a specific area and is then disbanded. The individual experts are listed with their affiliated organizations. However, this does not imply that the organization has approved the content of the standard or guideline. (Note: Per X9 policy, company names of non-member participants are listed only if, at time of publication, the X9 Secretariat received an original signed release permitting such company names to appear in print.)

The X9F1 Cryptographic Tool Standards and Guidelines group that developed this part of the Standard had the following members:

Miles Smid, Chairman  
Elaine Barker, Project Editor

### **Organization**

Certicom Corporation  
Communications Security Establishment of Canada  
Entrust  
HP  
Microsoft  
National Institute of Standards and Technology

### **Representative**

Dan Brown  
Bridget Walshe  
Don Johnson  
Susan Langford  
Niels Furguson  
Elaine Barker

**Draft ANS X9.82, Part 3 - November 2006**

National Security Agency

NTRU

Orion Security

Pitney Bowes, Inc

RSA Security

University Bank

Lily Chen

Morris Dworkin

John Kelsey

Paul Timmel

Michael Boyle

William Whyte

Miles Smid

Matt Compagna

James Randall

Steve Schmalz

Michael Talley

## **Random Number Generation**

### **Part 3: Deterministic Random Bit Generator Mechanisms**

#### **1 Scope**

The Standard consists of four parts:

- Part 1: Overview and Basic Principles
- Part 2: Entropy Sources
- Part 3: Deterministic Random Bit Generator Mechanisms
- Part 4: Random Bit Generator Construction

Part 1 should be read for a basic understanding of this Standard before reading Part 3. This part of ANSI X9.82 (Part 3) defines mechanisms for the generation of random bits using deterministic methods. The DRBG mechanisms are not sufficient by themselves to define a Random Bit Generator (RBG); Parts 2 and 4 of this Standard provide further requirements for the design of an RBG.

Part 3 includes:

1. A model for a deterministic random bit generator (DRBG),
2. Requirements for DRBG mechanisms,
3. Specifications for DRBG mechanisms that use hash functions, block ciphers and number theoretic problems,
4. Implementation issues, and
5. Assurance considerations.

A DRBG is based on a DRBG mechanism as specified in this part of the Standard and includes a source of entropy input. Part 3 specifies several diverse DRBG mechanisms, all of which provided acceptable security when this Standard was approved. However, in the event that new attacks are found on a particular class of mechanisms, a diversity of approved mechanisms will allow a timely transition to a different class of DRBG mechanism.

Random number generation does not require interoperability between two entities, e.g., communicating entities may use different DRBG mechanisms without affecting their ability to communicate. Therefore, an entity may choose a single appropriate DRBG mechanism for its applications; see Annex D for a discussion of DRBG selection.

The precise structure, design and development of a random bit generator is outside the scope of this Standard.

#### **2 Conformance**

An implementation of a DRBG mechanism may claim conformance with ANS X9.82 if it implements the mandatory provisions of Part 1 and the mandatory requirements of one or more of the DRBG mechanisms specified in this part of the Standard. An implementation of a DRBG may claim conformance with ANS X9.82 as an RBG if the following are implemented: the mandatory provisions of Part 1, the mandatory requirements of one or more of the DRBG mechanisms specified in this part of the Standard, an entropy source from Part 2 and the appropriate mandatory requirements of Part 4.

It is expected that conformance may be assured by a testing laboratory associated with the Cryptographic Module Validation Program (CMVP) (see <http://csrc.nist.gov/cryptval>). Although an implementation may claim conformance with the Standard apart from such testing, implementation testing through the CMVP is strongly recommended.

### **3 Normative References**

The following referenced documents are indispensable for the application of this Standard. For dated references, only the edition cited applies. Nevertheless, parties to agreements based on this document are encouraged to consider applying the most recent edition of the referenced documents indicated below. For undated references, the latest edition of the referenced document (including any amendments) applies.

ANS X9.52-1998, *Triple Data Encryption Algorithm Modes of Operation*.

ANS X9.62-2005, *Public Key Cryptography for the Financial Services Industry - The Elliptic Curve Digital Signature Algorithm (ECDSA)*.

ANS X9.63-2000, *Public Key Cryptography for the Financial Services Industry - Key Agreement and Key Transport Using Elliptic Key Cryptography*.

ANS X9.82, Part 1-200x, *Overview and Basic Principles*, Draft.

ANS X9.82, Part 2-200x, *Entropy Sources*, Draft.

ANS X9.82, Part 4-200x, *RBG Constructions*, Draft.

FIPS 140-2, *Security Requirements for Cryptographic Modules*; ASC X9 Registry 00001.

FIPS 180-2, *Secure Hash Standard (SHS)*, August 2002; ASC X9 Registry 00003.

FIPS 197, *Advanced Encryption Standard (AES)*, November 2001; ASC X9 Registry 00002.

FIPS 198, *Keyed-Hash Message Authentication Code (HMAC)*, March 6, 2002; ASC X9 Registry 00004.

### **4 Terms and Definitions**

Many of the terms used in Part 3 are defined in Part 1. Additional terms are defined in this section.

#### **4.1**

##### **Bitwise Exclusive-Or**

An operation on two bitstrings of equal length that combines corresponding bits of each bitstring using an exclusive-or operation.

#### **4.2**

##### **Conditioned Entropy Source**

An entropy source that either includes a conditioning function or for which conditioning is performed on the output of the entropy source. The conditioning function ensures that the conditioned entropy source provides full entropy bitstrings.

#### **4.3**

##### **Deterministic Random Bit Generator (DRBG)**

An RBG that includes a DRBG mechanism and a source of entropy input. The DRBG produces a sequence of bits from a secret initial value called a seed, along with other possible inputs. A DRBG is often called a Pseudorandom Number (or Bit) Generator.

#### 4.4

##### **DRBG Mechanism**

The portion of an RBG that includes the functions necessary to instantiate and uninstantiate the RBG, generate pseudorandom bits, (optionally) reseed the RBG and test the health of the the DRBG mechanism.

#### 4.5

##### **DRBG Mechanism Boundary**

A conceptual boundary that is used to explain the operations of a DRBG mechanism and its interaction with and relation to other processes.

#### 4.6

##### **Entropy**

A measure of the disorder, randomness or variability in a closed system. The entropy of  $X$  is a mathematical measure of the amount of information provided by an observation of  $X$ . As such, entropy is always relative to an observer and his or her knowledge prior to an observation. Also, see min-entropy.

#### 4.7

##### **Entropy Input**

The input to a DRBG mechanism of a string of bits that contains entropy; that is, the entropy input is digitized and is assessed.

#### 4.8

##### **Entropy Source**

A source of unpredictable data. There is no assumption that the unpredictable data has a uniform distribution. The entropy source includes a noise source, such as thermal noise or hard drive seek times; a digitization process; an assessment process; an optional conditioning process and health tests. Contrast with the Source of Entropy Input.

#### 4.9

##### **Exclusive-or**

A mathematical operation; the symbol  $\oplus$ , defined as:

$$\begin{aligned}0 \oplus 0 &= 0 \\0 \oplus 1 &= 1 \\1 \oplus 0 &= 1 \\1 \oplus 1 &= 0.\end{aligned}$$

Equivalent to binary addition without carry.

#### 4.10

##### **Hash Function**

A (mathematical) function that maps values from a large (possibly very large) domain into a smaller range. The function satisfies the following properties:

1. (One-way) It is computationally infeasible to find any input that maps to any pre-specified output;
2. (Collision free) It is computationally infeasible to find any two distinct inputs that map to the same output.

#### **4.11**

##### **Health Testing**

Testing within an implementation immediately prior to or during normal operation to determine that the implementation continues to perform as implemented and as validated (if implementation validation was performed).

#### **4.12**

##### **Implementation Testing for Validation**

Testing by an independent and accredited party to ensure that an implementation of this Standard conforms to the specifications of this Standard.

#### **4.13**

##### **Instantiation of an RBG**

An instantiation of an RBG is a specific, logically independent, initialized RBG. One instantiation is distinguished from another by a handle (e.g., an identifying number).

#### **4.14**

##### **Internal State**

The collection of stored information about a DRBG instantiation. This can include both secret and non-secret information.

#### **4.15**

##### **Personalization String**

An optional string of bits that is combined with an entropy input and (possibly) a nonce to produce a seed.

#### **4.16**

##### **Prediction Resistance**

Assurance that a compromise of the DRBG internal state has no effect on the security of future DRBG outputs. That is, an adversary who is given access to all of the output sequence after the compromise cannot distinguish it from random; if the adversary knows only part of the future output sequence, he cannot predict any bit of that future output sequence that he has not already seen. The complementary assurance is called Backtracking Resistance.

#### **4.17**

##### **Public Key Pair**

In an asymmetric (public) key cryptosystem, the public key and associated private key.

#### **4.18**

##### **Random Number Generator (RNG)**

A device or algorithm that outputs a sequence of binary bits that appears to be statistically independent and unbiased. An RBG is either a DRBG or an NRBG.

#### **4.19**

##### **Reseed**

To acquire additional bits with sufficient entropy for the desired security strength.

#### **4.20**

##### **Secure channel**

A path for transferring data between two entities or components that ensures confidentiality, integrity and replay protection, as well as mutual authentication between the entities or components. The secure channel may be provided using cryptographic, physical or procedural methods, or a combination thereof.

#### **4.21**

##### **Security Strength**

A number associated with the amount of work (that is, the number of operations) that is required to break a cryptographic algorithm or system; a security strength is specified in bits and is a specific value from the set (112, 128, 192, 256) for this Standard. The amount of work needed is  $2^{\text{security\_strength}}$ .

#### **4.22**

##### **Seed**

Noun : A string of bits that is used as input to a DRBG mechanism. The seed will determine a portion of the internal state of the DRBG, and its entropy must be sufficient to support the security strength of the DRBG.

Verb : To acquire bits with sufficient entropy for the desired security strength. These bits will be used as input to a DRBG mechanism to determine a portion of the initial internal state. Also see reseed.

#### **4.23**

##### **Seedlife**

The length of the seed period.

#### **4.24**

##### **Source of Entropy Input**

The source of the entropy input for a DRBG mechanism. Contrast with Entropy Source.

#### **4.25**

##### **Working State**

A subset of the internal state that is used by a DRBG mechanism to produce pseudorandom bits at a given point in time. The working state (and thus, the internal state) is updated to the next state prior to producing another string of pseudorandom bits.


## **5 Abbreviations and Symbols**

The following abbreviations are used in this document.

<b>Abbreviation</b>	<b>Meaning</b>
AES	Advanced Encryption Standard.
ANS	American National Standard
ASC	Accredited Standards Committee
DRBG	Deterministic Random Bit Generator.
ECDLP	Elliptic Curve Discrete Logarithm Problem.
FIPS	Federal Information Processing Standard.
HMAC	Keyed-Hash Message Authentication Code.
NRBG	Non-deterministic Random Bit Generator.

RBG	Random Bit Generator.
TDEA	Triple Data Encryption Algorithm.

The following symbols are used in this document.

Symbol	Meaning
$+$	Addition.
$\lceil X \rceil$	Ceiling: the smallest integer $\geq X$ . For example, $\lceil 5 \rceil = 5$ , and $\lceil 5.3 \rceil = 6$ .
$\lfloor X \rfloor$	Floor: The largest integer less than or equal to $X$ . For example, $\lfloor 5 \rfloor = 5$ , and $\lfloor 5.3 \rfloor = 5$ .
$X \oplus Y$	Bitwise exclusive-or (also bitwise addition modulo 2) of two bitstrings $X$ and $Y$ of the same length.
$X    Y$	Concatenation of two strings $X$ and $Y$ . $X$ and $Y$ are either both bitstrings, or both byte strings.
$\text{gcd}(x, y)$	The greatest common divisor of the integers $x$ and $y$ .
$\text{len}(a)$	The length in bits of string $a$ .
$x \bmod n$	The unique remainder $r$ (where $0 \leq r \leq n-1$ ) when integer $x$ is divided by $n$ . For example, $23 \bmod 7 = 2$ .
	Used in a figure to illustrate a "switch" between sources of input.
$\{a_1, \dots, a_i\}$	The internal state of the DRBG at a point in time. The types and number of the $a_i$ depends on the specific DRBG mechanism.
$0xab$	Hexadecimal notation that is used to define a byte (i.e., 8 bits) of information, where $a$ and $b$ each specify 4 bits of information and have values from the range $\{0, 1, 2, \dots, F\}$ . For example, $0xc6$ is used to represent 11000110, where $c$ is 1100, and 6 is 0110.
$0^x$	A string of $x$ zero bits.



## 6 General Discussion and Organization

Part 1 of this Standard (*Random Number Generation, Part 1: Overview and Basic Principles*) describes several cryptographic applications for random numbers and specifies the characteristics for random numbers and random number generators, introducing the concept of non-deterministic random bit generators (NRBGs) and deterministic random bit generators (DRBGs). In addition, Part 1 also introduces a general functional model and identifies the security properties expected for cryptographic random number generators.

Part 2 of this Standard (*Entropy Sources*) discusses entropy sources used by random bit generators. In the case of DRBGs, the entropy sources are required to obtain seeds for the DRBG.

Part 4 of this Standard (*Random Bit Generator Constructions*) provides guidance on combining components to construct secure random bit generators.

This part of the Standard (*Random Number Generation, Part 3: Deterministic Random Bit Generator Mechanisms*) specifies Approved DRBG mechanisms. A DRBG mechanism is an RBG component that utilizes an algorithm to produce a sequence of bits from an initial internal state that is determined by an input that is commonly known as a seed, which is constructed using entropy input. Because of the deterministic nature of the process, a DRBG mechanism is said to produce "pseudorandom" rather than random bits, i.e., the string of bits produced by a DRBG mechanism is predictable and can be reconstructed, given knowledge of the algorithm, the entropy input, the seed and any other input information. However, if the seed and entropy input are kept secret, and the algorithm is well designed, then the bitstrings will be unpredictable, up to the security strength provided by the DRBG.

The seed for a DRBG mechanism requires that sufficient entropy be provided during instantiation and reseeding (see Parts 2 and 4 of this Standard). While a DRBG mechanism may conform to this part of the Standard (i.e., Part 3), a DRBG cannot achieve the properties specified in Part 1 unless the source of entropy input is included as specified in Part 4. That is, the security of an RBG that uses a DRBG mechanism is a system implementation issue; both the DRBG mechanism and its source of entropy input must be considered.

The remaining sections of this part of the Standard are organized as follows:

- Section 7 provides a functional model for an RBG that uses a DRBG mechanism and discusses the major components of the DRBG mechanism.
- Section 8 provides concepts and general requirements for the implementation and use of a DRBG mechanism.
- Section 9 specifies the functions of a DRBG mechanism that are introduced in Section 8. These functions use the DRBG algorithms specified in Section 10.
- Section 10 specifies Approved DRBG algorithms.
- Section 11 addresses assurance issues for DRBG mechanisms.

This part of the Standard also includes the following normative annexes:

- Annex A specifies additional DRBG-specific information.
- Annex B provides conversion routines.

The following informative annexes are also included:

- Annex C discusses security considerations for selecting and implementing DRBG mechanisms.
- Annex D provides a discussion on DRBG mechanism selection.

- Annex E provides example pseudocode for each DRBG mechanism.
- Annex F relates the security properties identified in Part 1 to the requirements and specifications in Part 3.
- Annex G provides a bibliography for related informational material.

## 7 Functional Model

### 7.1 General Discussion

Figure 1 provides a functional model of an RBG (i.e., a DRBG) that is based on a DRBG mechanism. An RBG that uses a DRBG mechanism includes a source of entropy input and, depending on the implementation of the DRBG mechanism, includes a nonce source. The components of this model are discussed in the following subsections.

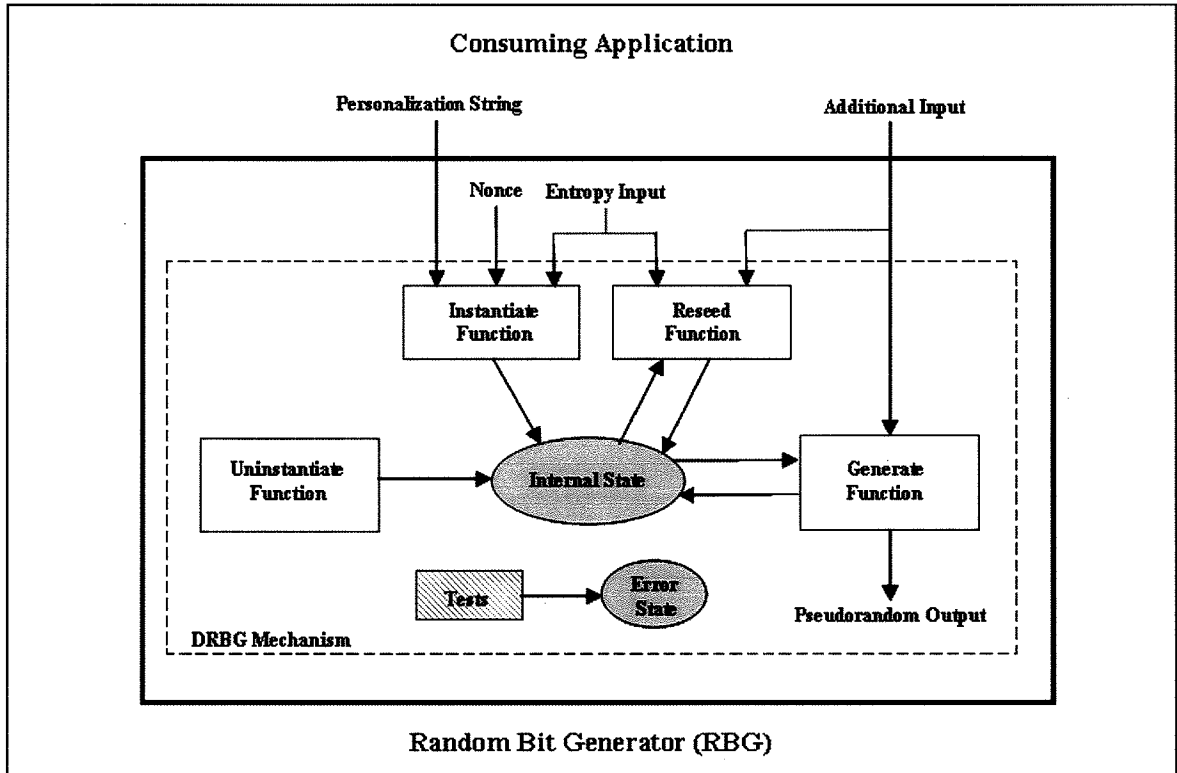


Figure 1: DRBG Functional Model

### 7.2 Functional Model Components

#### 7.2.1 Entropy Input

The entropy input is provided to a DRBG mechanism for the seed (see Section 8.4.2). The entropy input and the seed **shall** be kept secret. The secrecy of this information provides the basis for the security of the DRBG. At a minimum, the entropy input **shall** provide the amount of entropy requested by the DRBG mechanism. Appropriate sources for the entropy input are discussed in Parts 2 and 4 of this Standard.

Ideally, the entropy input will have full entropy; however, the DRBG mechanisms have been specified to allow for some bias in the entropy input by allowing the length of the entropy input to be longer than the required amount of entropy (expressed in bits). The entropy input can be defined to be a variable length (within limits), as well as fixed length. In all cases, the DRBG mechanism

expects that when entropy input is requested, the returned bitstring will contain at least the requested amount of entropy. Additional entropy beyond the amount requested is not required, but is desirable.

### **7.2.2 Other Inputs**

Other information may be obtained by a DRBG mechanism as input. This information may or may not be required to be kept secret by a consuming application; however, the security of the DRBG itself does not rely on the secrecy of this information. The information **should** be checked for validity when possible; for example, if time is used as an input, the format and reasonableness of the time could be checked.

During DRBG instantiation, a nonce may be required, and if used, it is combined with the entropy input to create the initial DRBG seed. The nonce and its use are discussed in Section 8.4.2.

This Standard recommends the insertion of a personalization string during DRBG instantiation; when used, the personalization string is combined with the entropy input bits and possibly a nonce to create the initial DRBG seed. The personalization string **shall** be unique for all instantiations of the same DRBG mechanism type (e.g., HMAC\_DRBG). See Section 8.5.2 for additional discussion on personalization strings.

Additional input may also be provided during reseeding and when pseudorandom bits are requested. See Section 8.5.3 for a discussion of this input.

### **7.2.3 The Internal State**

The internal state is the memory of the DRBG and consists of all of the parameters, variables and other stored values that the DRBG mechanism uses or acts upon. The internal state contains both administrative data (e.g., the security strength) and data that is acted upon and/or modified during the generation of pseudorandom bits (i.e., the *working state*).

### **7.2.4 The DRBG Mechanism Functions**

The DRBG mechanism functions handle the DRBG's internal state. The DRBG mechanisms in this Standard have five separate functions:

1. The instantiate function acquires entropy input and may combine it with a nonce and a personalization string to create a seed from which the initial internal state is created.
2. The generate function generates pseudorandom bits upon request, using the current internal state, and generates a new internal state for the next request.
3. The reseed function acquires new entropy input and combines it with the current internal state and any additional input that is provided to create a new seed and a new internal state.
4. The unstantiate function erases the internal state.
5. The health test function determines that the DRBG mechanism continues to function correctly.

## 8. DRBG Mechanism Concepts and General Requirements

### 8.1 Introduction

This section provides concepts and general requirements for the implementation and use of a DRBG mechanism. The DRBG mechanism functions are explained and requirements for an implementation are provided.

### 8.2 DRBG Mechanism Functions and a DRBG Instantiation

#### 8.2.1 DRBG Mechanism Functions

A DRBG mechanism requires instantiate, uninstantiate, generate, and health testing functions. A DRBG mechanism may also include a reseed function. A DRBG **shall** be instantiated prior to the generation of output by the DRBG. These functions are specified in Section 9.

#### 8.2.2 DRBG Instantiations

A DRBG may be used to obtain pseudorandom bits for different purposes (e.g., DSA private keys and AES keys) and may be separately instantiated for each purpose.

A DRBG is instantiated using a seed and may be reseeded; when reseeded, the seed **shall** be different than the seed used for instantiation. Each seed defines a *seed period* for the DRBG instantiation; an instantiation consists of one or more seed periods that begin when a new seed is acquired (see Figure 2).

#### 8.2.3 Internal States

During instantiation, an initial internal state is derived from the seed. The internal state for an instantiation includes:

1. Working state:
  - a. One or more values that are derived from the seed and become part of the internal state; these values **should** remain secret, and
  - b. A count of the number of requests or blocks produced since the instantiation was seeded or reseeded.
2. Administrative information (e.g., security strength and prediction resistance flag).

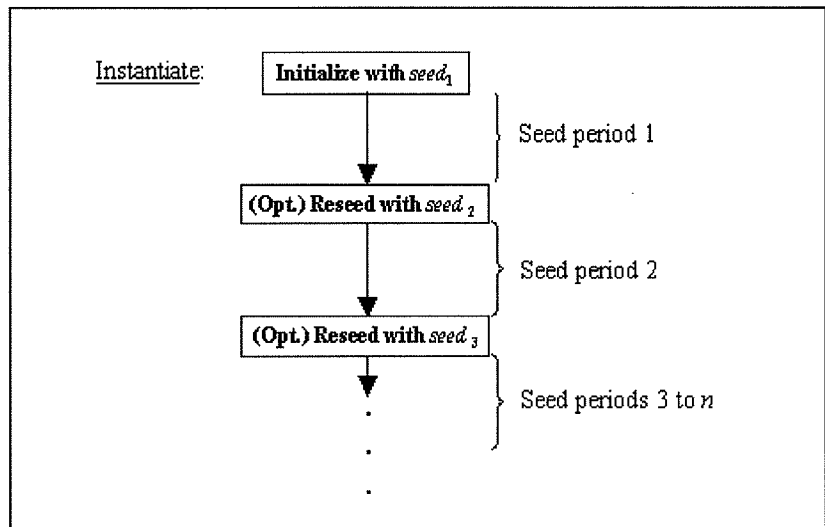


Figure 2: DRBG Instantiation

The internal state **shall** be protected at least as well as the intended use of the pseudorandom output bits requested by the consuming application. Each DRBG instantiation **shall** have its own internal state; the internal state for one DRBG instantiation **shall not** be used as the internal state for a different instantiation.

A DRBG transitions between internal states when the generator is requested to provide new pseudorandom bits. A DRBG may also be implemented to transition in response to internal or external events (e.g., system interrupts) or to transition continuously (e.g., whenever time is available to run the generator).

A DRBG mechanism implementation may be designed to handle multiple instantiations. Sufficient space must be available for the expected number of instantiations, i.e., sufficient memory must be available to store the internal state associated with each instantiation.

#### **8.2.4 Security Strengths Supported by an Instantiation**

The DRBG mechanisms specified in this Standard support four security strengths: 112, 128, 192 or 256 bits. A security strength for the instantiation is requested by a consuming application during instantiation, and the instantiate function obtains the appropriate amount of entropy for the requested security strength. Any security strength may be requested (up to a maximum of 256 bits), but the DRBG will only be instantiated to one of the four security strengths above, depending on the DRBG implementation. A requested security strength that is below the 112-bit security strength or is between two of the four security strengths will be instantiated to the next highest strength (e.g., a requested security strength of 80 bits will result in an instantiation at the 112-bit security strength).

The actual security strength supported by a given instantiation depends on the DRBG implementation and on the amount of entropy provided to the instantiate function in the entropy input. Note that the security strength actually supported by a particular instantiation could be less than the maximum security strength possible for that DRBG implementation (see Table 1). For example, a DRBG that is designed to support a maximum security strength of 256 bits could be instantiated to support only a 128-bit security strength if the additional security provided by the 256-bit security strength is not required (i.e., by requesting only 128 bits of entropy during instantiation, rather than 256 bits of entropy).

**Table 1: Possible Instantiated Security Strengths**

<b>Maximum Designed Security Strength</b>	<b>112</b>	<b>128</b>	<b>192</b>	<b>256</b>
<b>Possible Instantiated Security Strengths</b>	112	112, 128	112, 128, 192	112, 128, 192, 256

Following instantiation, requests can be made to the generate function for pseudorandom bits. For each generate request, a security strength to be provided for the bits is requested. Any security strength can be requested during a call to the generate function, up to the security strength of the instantiation, e.g., an instantiation could be instantiated at the 128-bit security strength, but a request for pseudorandom bits could indicate that a lesser security strength is actually required for the bits to be generated. The generate function checks that the requested security strength does not exceed the security strength for the instantiation. Assuming that the request is valid, the requested number of bits is returned.

When an instantiation is used for multiple purposes, the minimum entropy requirement for each purpose must be considered. The DRBG needs to be instantiated for the highest security strength required. For example, if one purpose requires a security strength of 112 bits, and another purpose

requires a security strength of 256 bits, then the DRBG needs to be instantiated to support the 256-bit security strength.

### 8.3 DRBG Mechanism Boundaries

As a convenience, this Standard uses the notion of a “DRBG mechanism boundary” to explain the operations of a DRBG mechanism and its interaction with and relation to other processes; a DRBG mechanism boundary contains all DRBG mechanism functions and internal states required for a DRBG. Data enters a DRBG mechanism boundary via the DRBG’s public interfaces, which are made available to consuming applications.

Within a DRBG mechanism boundary,

1. The DRBG internal state and the operation of the DRBG mechanism functions **shall** only be affected according to the DRBG mechanism specification.
2. The DRBG internal state **shall** exist solely within the DRBG mechanism boundary. The internal state **shall** be contained within the DRBG mechanism boundary and **shall not** be accessed by non-DRBG functions or other instantiations of that DRBG or other DRBGs.
3. Information about secret parts of the DRBG internal state and intermediate values in computations involving these secret parts **shall not** affect any information that leaves the DRBG mechanism boundary, except as specified for the DRBG pseudorandom bit outputs.

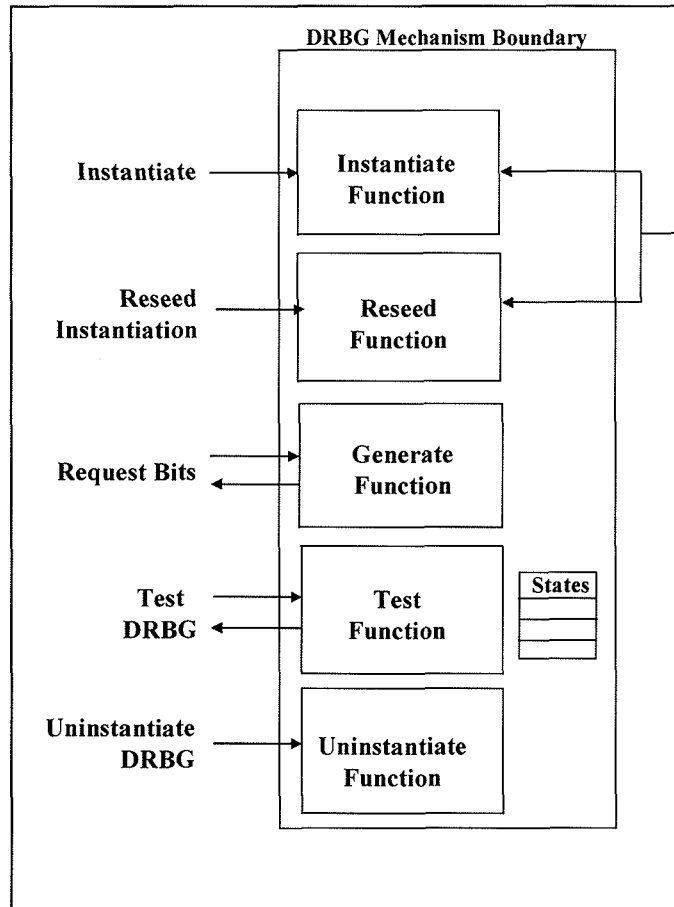


Figure 3: DRBG Mechanism Functions Within a Single Device

Each DRBG mechanism includes one or more cryptographic primitives (e.g., a hash function). Other applications may use the same cryptographic primitive as long as the DRBG’s internal state and the DRBG mechanism functions are not affected.

A DRBG mechanism’s functions may be contained within a single device, or may be distributed across multiple devices (see Figures 3 and 4). Figure 3 depicts a DRBG for which all functions are contained within the same device. Figure 4 provides an example of DRBG mechanism functions that are distributed across multiple devices. In this latter case, each device has a DRBG mechanism sub-boundary that contains the DRBG mechanism functions implemented on that device. The boundary around the entire DRBG mechanism **shall** include the aggregation of sub-boundaries providing the DRBG mechanism functionality. The use of distributed DRBG mechanism

functions may be convenient for restricted environments (e.g., smart card applications) in which the primary use of the DRBG does not require repeated use of the instantiate or reseed functions.

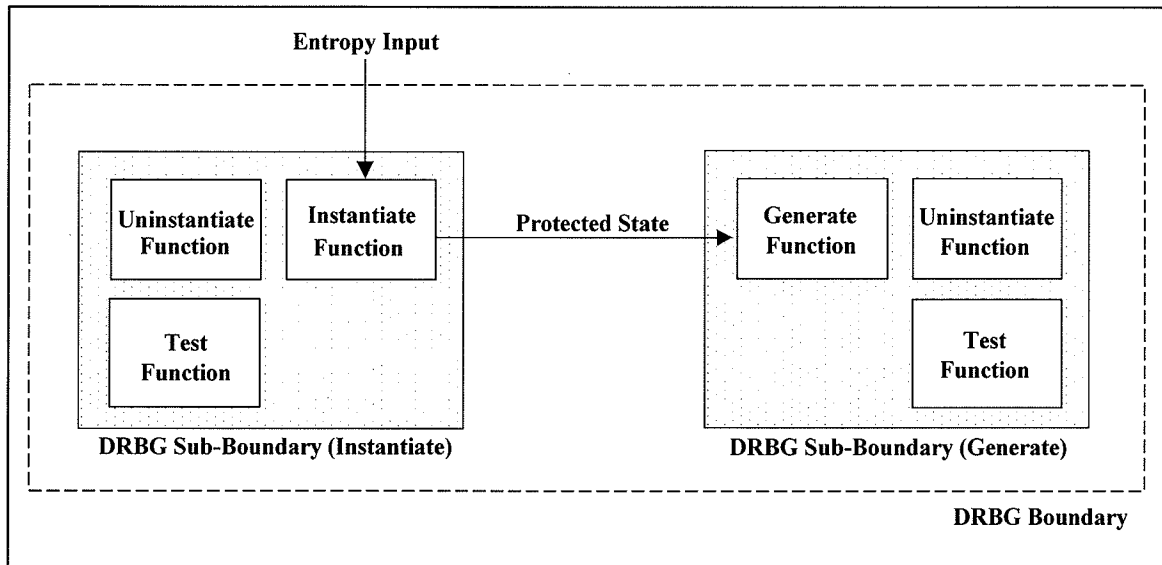


Figure 4: Distributed DRBG Functions

Each DRBG mechanism boundary or sub-boundary **shall** contain a test function to test the “health” of other DRBG mechanism functions within that boundary. In addition, each boundary or sub-boundary **shall** contain an uninstantiate function in order to perform and/or react to health testing.

When DRBG mechanism functions are distributed, a secure channel **shall** be used to protect the internal state or parts of the internal state that are transferred between the distributed DRBG mechanism sub-boundaries. The security provided by the secure channel **shall** be consistent with the security required by the consuming application.

## 8.4 Seeds

### 8.4.1 General Discussion

When a DRBG is used to generate pseudorandom bits, entropy input is acquired in order to generate a seed prior to the generation of output bits by the DRBG. The seed is used to instantiate the DRBG and determine the initial internal state that is used when calling the DRBG to obtain the first output bits.

Reseeding is a means of restoring the secrecy of future outputs of the DRBG if a seed or the internal state becomes known. Periodic reseeding is a good way of addressing the threat of the DRBG seed, entropy input or working state being compromised over time. In some implementations (e.g., smartcards), an adequate reseeding process may not be possible. In these cases, the best policy might be to replace the DRBG, obtaining a new seed in the process (e.g., obtain a new smart card).

### 8.4.2 Generation and Handling of Seeds



The seed and its use by a DRBG mechanism is generated and handled as follows:

1. Seed construction for instantiation: Figure 5 depicts the seed construction process for instantiation. The seed material used to determine a seed for instantiation consists of entropy input, a nonce and an optional personalization string. Entropy input is always used in the construction of a seed; requirements for the entropy input are discussed in item 3. Except for the case noted below, a nonce is used; requirements for the nonce are

discussed in item 7. A personalization string **should** also be used; requirements for the personalization string are discussed in Section 8.5.2.

Depending on the DRBG mechanism and the source of the entropy input, a derivation function may be required to derive a seed from the seed material. However, in certain circumstances, the DRBG mechanism based on block cipher algorithms (see Section 10.3) may be implemented without a derivation function. When implemented in this manner, a (separate) nonce (as shown in Figure 5) is not used. Note, however, that the personalization string could contain a nonce, if desired.

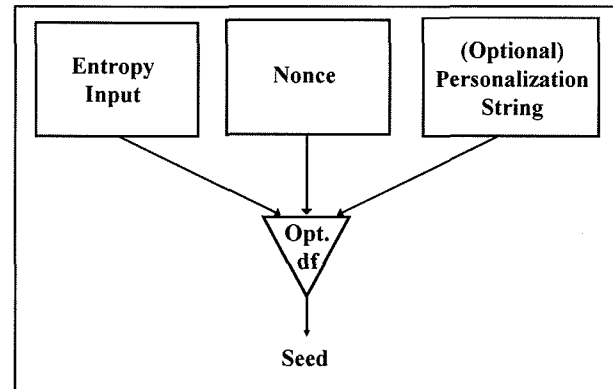


Figure 5: Seed Construction for Instantiation

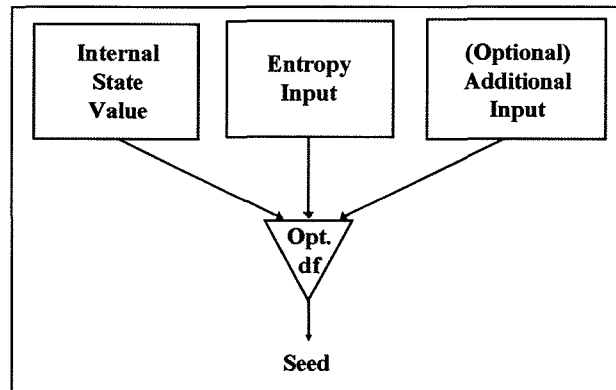


Figure 6: Seed Construction for Reseeding

2. Seed construction for reseeding: Figure 6 depicts the seed construction process for reseeding an instantiation. The seed material for reseeding consists of a value that is carried in the internal state<sup>1</sup>, new entropy input and, optionally, additional input. The internal state value and the entropy input are required; requirements for the entropy input are discussed in item 3. Requirements for the additional input are discussed in Section 8.5.3. As in item 1, a derivation function may be required for reseeding. See item 1 for further guidance.
3. Entropy requirements for the entropy input: The entropy input **shall** have entropy that is equal to or greater than the security strength of the instantiation. Additional entropy may be provided in the nonce or the optional personalization string during instantiation, or in the additional input during reseeding and generation, but this is not required. The use of more entropy than the minimum value will offer a security "cushion". This may be useful if the

<sup>1</sup> See each DRBG mechanism specification for the value that is used.

assessment of the entropy provided in the entropy input is incorrect. Having more entropy than the assessed amount is acceptable; having less entropy than the assessed amount could be fatal to security. The presence of more entropy than is required, especially during the instantiation, will provide a higher level of assurance than the minimum required entropy.

4. Seed length: The minimum length of the seed depends on the DRBG mechanism and the security strength required by the consuming application. See Section 10.
5. Source of entropy input: The source of the entropy input **shall** be either:
  - a. An Approved NRBG,
  - b. An Approved DRBG, thus forming a chain of at least two DRBGs; the highest-level DRBG in the chain **shall** be seeded by an Approved NRBG or an entropy source, or
  - c. An appropriate entropy source.

Further discussion about the source of entropy input is provided in Parts 2 and 4 of this Standard.

6. Entropy input and seed privacy: The entropy input and the resulting seed **shall** be handled in a manner that is consistent with the security required for the data protected by the consuming application. For example, if the DRBG is used to generate keys, then the entropy inputs and seeds used to generate the keys **shall** (at a minimum) be protected as well as the key.
7. Nonce: A nonce may be required in the construction of a seed during instantiation in order to provide a security cushion to block certain attacks. The nonce **shall** be either:
  - a. A value with at least  $(1/2 \text{ security\_strength})$  bits of entropy,
  - b. A value that is expected to repeat no more often than a  $(1/2 \text{ security\_strength})$ -bit random string would be expected to repeat.

For case a, the nonce may be acquired from the same source and at the same time as the entropy input. In this case, the seed could be considered to be constructed from an "extra strong" entropy input and the optional personalization string, where the entropy for the entropy input is equal to or greater than  $(3/2 \text{ security\_strength})$  bits.

The nonce ensures that the DRBG provides *security\_strength* bits of security to the consuming application. When a DRBG is instantiated many times without a nonce, a compromise may become more likely. In some consuming applications, a single DRBG compromise may reveal long-term secrets (e.g., a compromise of the DSA per-message secret reveals the signing key).

8. Reseeding: 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 data that is protected by cryptographic mechanisms that use the DRBG.

Seeds have a finite seedlife (i.e., the number of blocks or outputs that are produced during a seed period); the maximum seedlife is dependent on the DRBG mechanism used. Reseeding is accomplished by 1) an explicit reseeding of the DRBG by the consuming application, or 2) by the generate function when either prediction resistance is requested, or when the limit of the seedlife is reached.

Reseeding of the DRBG **shall** be performed in accordance with the specification for the given DRBG mechanism. The DRBG reseed specifications within this Standard are

designed to produce a new seed that is determined by both the current internal state and newly-obtained entropy input that will support the desired security strength.

An alternative to reseeding is to create an entirely new instantiation. However, reseeding is preferred over creating a new instantiation. If a DRBG instantiation was initially seeded with sufficient entropy, and the source of entropy input subsequently fails without being detected, then a new instantiation using the same (failed) source of entropy input would not have sufficient entropy to operate securely. However, if there is an undetected failure in the source of entropy input for an already properly seeded DRBG instantiation, the DRBG instantiation will still retain any previous entropy when the reseed operation fails to introduce new entropy.

9. Seed use: The seed that is used to initialize one instantiation of a DRBG **shall not** be intentionally used to reseed the same instantiation or used as a seed for another DRBG instantiation. Note that a DRBG does not provide output until a seed is available, and the internal state has been initialized.
10. Entropy input and seed separation: The seed used by DRBG and the entropy input used to create that seed **shall not** intentionally be used for other purposes (e.g., domain parameter or prime number generation).

## 8.5 Other Inputs to the DRBG Mechanism

### 8.5.1 Discussion

Other input may be provided during DRBG instantiation, pseudorandom bit generation and reseeding. This input may contain entropy, but this is not required. During instantiation, a personalization string may be provided and combined with entropy input and a nonce to derive a seed (see Section 8.5.2). When pseudorandom bits are requested and when reseeding is performed, additional input may be provided (see Section 8.5.3).

Depending on the method for acquiring the input, the exact value of the input may or may not be known to the user or consuming application. For example, the input could be derived directly from values entered by the user or consuming application, or the input could be derived from information introduced by the user or consuming application (e.g., from timing statistics based on key strokes or movements of the computer's mouse), or the input could be the output of another RBG.

### 8.5.2 Personalization String

During instantiation, a personalization string **should** be used to derive the seed (see Section 8.4.2). The intent of a personalization string is to differentiate this DRBG instantiation from all other instantiations that might ever be created. The personalization string **should** be set to some bitstring that is as unique as possible, and may include secret information. Secret information **should not** be used in the personalization string if it requires a level of protection that is greater than the intended security strength of the DRBG instantiation. Good choices for the personalization string contents include:

- Device serial numbers,
- Public keys,
- User identification,
- Secret per-module or per-device values,
- Timestamps,
- Network addresses,
- Special secret key values for this specific DRBG instantiation,
- Application identifiers,
- Protocol version identifiers,
- Random numbers,

- Seedfiles,
- Nonces.

### **8.5.3 Additional Input**

During each request for bits from a DRBG and during reseeding, the insertion of additional input is allowed. This input is optional, and the ability to enter additional input may or may not be included in an implementation. Additional input may be restricted, depending on the implementation and the DRBG mechanism. The use of additional input may be a means of providing more entropy for the DRBG internal state that will increase assurance that the entropy requirements are met. If the additional input is kept secret and has sufficient entropy, the input can provide more assurance when recovering from the compromise of the entropy input, the seed or one or more DRBG internal states.

### **8.6 Prediction Resistance and Backtracking Resistance**

Part 1 discusses backtracking and prediction resistance. All DRBGs in this Standard have been designed to provide backtracking resistance within an instantiation. Prediction resistance can be provided only by ensuring that a DRBG is effectively reseeded between DRBG requests. The DRBG mechanisms in this Standard can (optionally) be implemented to support prediction resistance (see Section 9), and a user or application can request prediction resistance when needed.

## 9 DRBG Mechanism Functions

### 9.1 General Discussion

The DRBG mechanism functions in this Standard are specified as an algorithm (see Section 10) and an “envelope” of pseudocode around that algorithm (defined in this section). The pseudocode in the envelopes checks the input parameters, obtains input not provided by the input parameters, accesses the appropriate DRBG algorithm and handles the internal state. A function need not be implemented using such envelopes (e.g., all code may be implemented in-line), but the function **shall** have equivalent functionality.

During instantiation and reseeding (see Sections 9.2 and 9.3), entropy input is acquired for constructing a seed as discussed in Section 8.4.2. In the specifications of this Standard, a **Get\_entropy\_input** pseudo-function is used for this purpose. The entropy input **shall not** be provided by a consuming application as an input parameter in an instantiate or reseed request. The **Get\_entropy\_input** function is not fully specified in this Standard, but has the following meaning:

**Get\_entropy\_input**: A function that is used to obtain entropy input. The function call is:

$(status, entropy\_input) = \text{Get\_entropy\_input}(min\_entropy, min\_length, max\_length),$

which requests a string of bits (*entropy\_input*) with at least *min\_entropy* bits of entropy. The length for the string **shall** be equal to or greater than *min\_length* bits, and less than or equal to *max\_length* bits. A *status* code is also returned from the function.

Note that an implementation may choose to define this functionality differently; for example, for many of the DRBG mechanisms, the *min\_length* = *min\_entropy* for the **Get\_entropy\_input** function, in which case, the second parameter could be omitted.

In the pseudocode in this section, two classes of error codes are returned: **ERROR\_FLAG** and **CATASTROPHIC\_ERROR\_FLAG**. These error codes are discussed in Section 11.4.7.

Comments are often included in the pseudocode in this Standard. A comment placed on a line that includes pseudocode applies to that line; a comment placed on a line containing no pseudocode applies to one or more lines of pseudocode immediately below that comment.

### 9.2 Instantiating a DRBG

A DRBG **shall** be instantiated prior to the generation of pseudorandom bits. The instantiate function:

1. Checks the validity of the other input parameters,
2. Determines the security strength for the DRBG instantiation,
3. Determines any DRBG mechanism specific parameters (e.g., elliptic curve domain parameters),
4. Obtains entropy input with entropy sufficient to support the security strength,
5. Obtains the nonce (if required),
6. Determines the initial internal state using the instantiate algorithm,

7. If an implementation supports multiple simultaneous instantiations of the same DRBG, a *state\_handle* for the internal state is returned to the consuming application (see below).

Let *working\_state* be the working state for the particular DRBG mechanism, and let *min\_length*, *max\_length*, and *highest\_supported\_security\_strength* be defined for each DRBG mechanism (see Section 10). Let **Instantiate\_algorithm** be a call to the appropriate instantiate algorithm for the DRBG mechanism (see Section 10).

The following or an equivalent process **shall** be used to instantiate a DRBG.

**Instantiate\_function** (*requested\_instantiation*, *security\_strength*, *prediction\_resistance\_flag*, *personalization\_string*):

1. *requested\_instantiation\_security\_strength*: A requested security strength for the instantiation. Implementations that support only one security strength do not require this parameter; however, any application using that implementation must be aware of the security strength that is supported.
2. *prediction\_resistance\_flag*: Indicates whether or not prediction resistance may be required by a the consuming application during one or more requests for pseudorandom bits. Implementations that always provide or never support prediction resistance do not require this parameter. However, the user of a consuming application must determine whether or not prediction resistance may be required by the consuming application before electing to use such an implementation. If the *prediction\_resistance\_flag* is not needed (i.e., because prediction resistance is always performed or is not supported), then the *prediction\_resistance\_flag* input parameter and instantiate process step 2 are omitted, and the *prediction\_resistance\_flag* is omitted from the internal state in step 11 of the instantiate process.
3. *personalization\_string*: An optional input that provides personalization information (see Sections 8.4.2 and 8.5.2). The maximum length of the personalization string (*max\_personalization\_string\_length*) is implementation dependent, but **shall** be less than or equal to the maximum length specified for the given DRBG mechanism (see Section 10). If the input of a personalization string is not supported, then the *personalization\_string* input parameter and step 3 of the instantiate process are omitted, and instantiate process step 9 is modified to omit the personalization string.

**Required information not provided by the consuming application during instantiation:** This input **shall not** be provided by the consuming application as an input parameter during the instantiate request.

1. *entropy\_input*: Input bits containing entropy. The maximum length of the *entropy\_input* is implementation dependent, but **shall** be less than or equal to the specified maximum length for the selected DRBG mechanism (see Section 10).
2. *nonce*: A nonce as specified in Section 8.4.2. Note that if a random value is used as the nonce, the *entropy\_input* and *nonce* could be acquired using a single **Get\_entropy\_input** call (see step 6 of the instantiate process); in this case, the first parameter of the **Get\_entropy\_input** call is adjusted to include the entropy for the *nonce* (i.e., the *security\_strength* would be increased by at least  $\frac{1}{2}$  *security\_strength*), process step 8 is omitted, and the *nonce* is omitted from the parameter list in process step 9.

Note that in some cases, a nonce will not be used by a DRBG mechanism; in this case, step 8 is omitted, and the *nonce* is omitted from the parameter list in instantiate process step 9.

**Output to a consuming application after instantiation:**

1. *status*: The status returned from the instantiate function. The *status* will indicate **SUCCESS** or an **ERROR**. If an **ERROR** is indicated, either no *state\_handle* or an invalid *state\_handle* **shall** be returned. A consuming application **should** check the *status* to determine that the DRBG has been correctly instantiated.
2. *state\_handle*: Used to identify the internal state for this instantiation in subsequent calls to the generate, reseed, uninstantiate and test functions.

If a state handle is not required for an implementation because the implementation does not support multiple simultaneous instantiations, a *state\_handle* need not be returned. In this case, instantiate process step 10 is omitted, process step 11 is revised to save the only internal state, and process step 12 is altered to omit the *state\_handle*.

**Information retained within the DRBG mechanism boundary after instantiation:**

The internal state for the DRBG, including the *working\_state* and administrative information (see Sections 8.2.3 and 10 for definitions of the *working\_state* and administrative information).

**Instantiate Process:**

Comment: Check the validity of the input parameters.

1. If *requested\_instantiation\_security\_strength* > *highest\_supported\_security\_strength*, then return an **ERROR\_FLAG**.
2. If *prediction\_resistance\_flag* is set, and prediction resistance is not supported, then return an **ERROR\_FLAG**.
3. If the length of the *personalization\_string* > *max\_personalization\_string\_length*, return an **ERROR\_FLAG**.
4. Set *security\_strength* to the nearest security strength greater than or equal to *requested\_instantiation\_security\_strength*.

Comment: The following step is required by the **Dual\_EC\_DRBG** when multiple curves are available (see Section 10.4.2.2.2). Otherwise, the step is omitted.

5. Using the *security\_strength*, select appropriate DRBG mechanism parameters.

Comment: Obtain the entropy input.

6. (*status*, *entropy\_input*) = **Get\_entropy\_input** (*security\_strength*, *min\_length*, *max\_length*).
7. If an **ERROR** is returned in step 6, return a **CATASTROPHIC\_ERROR\_FLAG**.

8. Obtain a *nonce*.

Comment: This step **shall** include any appropriate checks on the acceptability of the *nonce*. See Section 8.4.2.

Comment: Call the appropriate instantiate algorithm in Section 10 to obtain values for the initial *working\_state*.

9. *initial\_working\_state* = **Instantiate\_algorithm** (*entropy\_input*, *nonce*, *personalization\_string*).
10. Get a *state\_handle* for a currently empty state. If an empty internal state cannot be found, return an **ERROR\_FLAG**.
11. Set the internal state indicated by *state\_handle* to the initial values for the internal state (i.e., set the *working\_state* to the values returned as *initial\_working\_state* in step 9 and any other values required for the *working\_state* (see Section 10), and set the administrative information to the appropriate values (e.g., the values of *security\_strength* and the *prediction\_resistance\_flag*).
12. Return **SUCCESS** and *state\_handle*.

### 9.3 Reseeding a DRBG Instantiation

The reseed of an instantiation is not required, but is recommended whenever a consuming application and implementation are able to perform this process. Reseeding will insert additional entropy into the generation of pseudorandom bits. Reseeding may be:

- explicitly requested by a consuming application,
- performed when prediction resistance is requested by a consuming application,
- triggered by the generate function when a predetermined number of pseudorandom outputs have been produced or a predetermined number of generate requests have been made (i.e., at the end of the seedlife), or
- triggered by external events (e.g., whenever sufficient entropy is available).

If a reseed capability is not supported, a new DRBG instantiation may be created (see Section 9.2).

The reseed function:

1. Checks the validity of the input parameters,
2. Obtains entropy input with sufficient entropy to support the security strength, and
3. Using the reseed algorithm, combines the current internal state with the new entropy input and any additional input to determine the new internal state.

Let *working\_state* be the working state for the particular DRBG instantiation, let *min\_length* and *max\_length* be defined for each DRBG mechanism, and let **Reseed\_algorithm** be a call to the appropriate reseed algorithm for the DRBG mechanism (see Section 10).

The following or an equivalent process **shall** be used to reseed the DRBG instantiation.

**Reseed\_function** (*state\_handle*, *additional\_input*):



1. *state\_handle*: A pointer or index that indicates the internal state to be reseeded. If a state handle is not used by an implementation because the implementation does not support multiple simultaneous instantiations, a *state\_handle* is not provided as input. Since there is only a single internal state in this case, reseed process step 1 obtains the contents of the internal state, and process step 6 replaces the *working\_state* of this internal state.
2. *additional\_input*: An optional input. The maximum length of the *additional\_input* (*max\_additional\_input\_length*) is implementation dependent, but **shall** be less than or equal to the maximum value specified for the given DRBG mechanism (see Section 10). If the input by a consuming application of *additional\_input* is not supported, then the input parameter and step 2 of the reseed process are omitted, and step 5 of the reseed process is modified to remove the *additional\_input* from the parameter list.

**Required information not provided by the consuming application during reseeding:**

1. *entropy\_input*: Input bits containing entropy. The maximum length of the *entropy\_input* is implementation dependent, but **shall** be less than or equal to the specified maximum length for the selected DRBG mechanism (see Section 10).
2. Internal state values required by the DRBG for reseeding, i.e., the *working\_state* and administrative information, as appropriate.

**Output to a consuming application after reseeding:**

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS** or an **ERROR**.

**Information retained within the DRBG mechanism boundary after reseeding:**

Replaced internal state values (i.e., the *working\_state*).

**Reseed Process:**

Comment: Get the current internal state and check the input parameters.

1. Using *state\_handle*, obtain the current internal state. If *state\_handle* indicates an invalid or empty internal state, return an **ERROR\_FLAG**.
2. If the length of the *additional\_input* > *max\_additional\_input\_length*, return an **ERROR\_FLAG**.

Comment: Obtain the entropy input.

3. (*status*, *entropy\_input*) = **Get\_entropy\_input** (*security\_strength*, *min\_length*, *max\_length*).
4. If an **ERROR** is returned in step 3, return a **CATASTROPHIC\_ERROR\_FLAG**.

Comment: Get the new *working\_state* using the appropriate reseed algorithm in Section 10.

5. *new\_working\_state* = **Reseed\_algorithm** (*working\_state*, *entropy\_input*, *additional\_input*).

Comment: Save the new values of the internal state.

6. Replace the *working\_state* in the internal state indicated by *state\_handle* with the values of *new\_working\_state* obtained in step 5.
7. Return SUCCESS.

#### 9.4 Generating Pseudorandom Bits Using a DRBG

This function is used to generate pseudorandom bits after instantiation or reseeding (see Sections 9.2 and 9.3). The generate function:

1. Checks the validity of the input parameters,
2. Calls the reseed function to obtain sufficient entropy if the instantiation needs additional entropy because the end of the seedlife has been reached or prediction resistance is required; see Sections 9.4.2 and 9.4.3 for more information on reseeding at the end of the seedlife and on handling prediction resistance requests.
3. Generates the requested pseudorandom bits using the generate algorithm. The generate algorithm will check that two consecutive outputs are not the same.
4. Updates the working state.
5. Returns the requested pseudorandom bits to the consuming application.

##### 9.4.1 The Generate Function

Let *outlen* be the length of the output block of the cryptographic primitive (see Section 10). Let **Generate\_algorithm** be a call to the appropriate generate algorithm for the DRBG mechanism (see Section 10), and let **Reseed\_function** be a call to the reseed function in Section 9.2.

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

**Generate\_function** (*state\_handle*, *requested\_number\_of\_bits*, *requested\_security\_strength*, *prediction\_resistance\_request*, *additional\_input*):

1. *state\_handle*: A pointer or index that indicates the internal state to be used. If a state handle is not used by an implementation because the implementation does not support multiple simultaneous instantiations, a *state\_handle* is not provided as input. The *state\_handle* is omitted from the input parameter list in process step 7.1, generate process steps 1 and 7.3 are used to obtain the contents of the internal state, and process step 10 replaces the *working\_state* of this internal state.
2. *requested\_number\_of\_bits*: The number of pseudorandom bits to be returned from the generate function. The *max\_number\_of\_bits\_per\_request* is implementation dependent but **shall** be less than or equal to the value provided in Section 10 for a specific DRBG mechanism.
3. *requested\_security\_strength*: The security strength to be associated with the requested pseudorandom bits. DRBG implementations that support only one security strength do not require this parameter; however, any consuming application using that DRBG implementation must be aware of the supported security strength.
4. *prediction\_resistance\_request*: Indicates whether or not prediction resistance is to be provided during the request. DRBGs that are implemented to always support prediction

resistance or that support prediction resistance do not require this parameter. However, when prediction resistance is not supported, the user of a consuming application must determine whether or not prediction resistance may be required by the application before electing to use such a DRBG implementation.

If prediction resistance is not supported, then the *prediction\_resistance\_request* input parameter and step 5 of the generate process is omitted, and generate process step 7 is modified to omit the check for the *prediction\_resistance\_request*.

If prediction resistance is always performed, then the *prediction\_resistance\_request* input parameter and generate process step 5 may be omitted, and generate process steps 7 and 8 are replaced by:

*status* = **Reseed\_function** (*state\_handle*, *additional\_input*).

If *status* indicates an **ERROR**, then return *status*.

Using *state\_handle*, obtain the new internal state.

(*status*, *pseudorandom\_bits*, *new\_working\_state*) = **Generate\_algorithm**  
(*working\_state*, *requested\_number\_of\_bits*).

Note that if the input of *additional\_input* is not supported, then the *additional\_input* parameter in the Reseed call above may be omitted.

5. *additional\_input*: An optional input. The maximum length of the *additional\_input* (*max\_additional\_input\_length*) is implementation dependent, but **shall** be less than or equal to the specified maximum length for the selected DRBG mechanism (see Section 10). If the input of *additional\_input* is not supported, then the input parameter, generate process steps 4 and 7.4 and the *additional\_input* input parameter in steps 7.1 and 8 are omitted.

**Required information not provided by the consuming application during generation:**

1. Internal state values required for generation for the *working\_state* and administrative information, as appropriate.

**Output to a consuming application after generation:**

1. *status*: The status returned from the generate function. The *status* will indicate **SUCCESS** or an **ERROR**.
2. *pseudorandom\_bits*: The pseudorandom bits that were requested.

**Information retained within the DRBG mechanism boundary after generation:**

Replaced internal state values (i.e., the new *working\_state*).

**Generate Process:**

Comment Get the internal state and check the input parameters.

1. Using *state\_handle*, obtain the current internal state for the instantiation. If *state\_handle* indicates an invalid or empty internal state, then return an **ERROR\_FLAG**.

2. If *requested\_number\_of\_bits* > *max\_number\_of\_bits\_per\_request*, then return an **ERROR\_FLAG**.
3. If *requested\_security\_strength* > the *security\_strength* indicated in the internal state, then return an **ERROR\_FLAG**.
4. If the length of the *additional\_input* > *max\_additional\_input\_length*, then return an **ERROR\_FLAG**.
5. If *prediction\_resistance\_request* is set, and *prediction\_resistance\_flag* is not set, then return an **ERROR\_FLAG**.
6. Clear the *reseed\_required\_flag*.      Comment: See Section 9.4.2 for discussion.  
  Comment: Reseed if necessary (see Section 9.3).
7. If *reseed\_required\_flag* is set, or if *prediction\_resistance\_request* is set, then
  - 7.1 *status* = **Reseed\_function** (*state\_handle*, *additional\_input*).
  - 7.2 If *status* indicates an **ERROR**, then return *status*.
  - 7.3 Using *state\_handle*, obtain the new internal state.
  - 7.4 *additional\_input* = the Null string.
  - 7.5 Clear the *reseed\_required\_flag*.  
  
  Comment: Request the generation of  
  *pseudorandom\_bits* using the appropriate  
  generate algorithm in Section 10.
8. (*status*, *pseudorandom\_bits*, *new\_working\_state*) = **Generate\_algorithm** (*working\_state*, *requested\_number\_of\_bits*, *additional\_input*).
9. If *status* indicates that a reseed is required before the requested bits can be generated, then
  - 9.1 Set the *reseed\_required\_flag*.
  - 9.2 Go to step 7.
10. Replace the old *working\_state* in the internal state indicated by *state\_handle* with the values of *new\_working\_state*.
11. Return **SUCCESS** and *pseudorandom\_bits*.

### Implementation notes:

If a reseed capability is not supported, or a reseed is not desired, then generate process steps 6 and 7 are removed; and step 9 is replaced by:

9. If *status* indicates that a reseed is required before the requested bits can be generated, then
  - 9.1 *status* = **Uninstantiate function** (*state handle*).

9.2 Return an indication that the DRBG instantiation can no longer be used.

#### 9.4.2 Reseeding at the End of the Seedlife

When pseudorandom bits are requested by a consuming application, the generate function checks whether or not a reseed is required by comparing the counter within the internal state (see Section 8.2.3) against a predetermined reseed interval for the DRBG implementation. This is specified in the generate process (see Section 9.4.1) as follows:

- a. Step 6 clears the *reseed\_required\_flag*.
- b. Step 7 checks the value of the *reseed\_required\_flag*. At this time, the *reseed\_required\_flag* is clear, so step 7 is skipped unless prediction resistance was requested by the consuming application. For the purposes of this explanation, assume that prediction resistance was not requested.
- c. Step 8 calls the **Generate\_algorithm**, which checks whether a reseed is required. If it is required, an appropriate *status* will be returned.
- d. Step 9 checks the *status* returned by the **Generate\_algorithm**. If the *status* indicates that a reseed is not required, the generate process continues with step 10.
- e. However, if the status indicates that a reseed is required, then the *reseed\_required\_flag* is set, and processing continues by going back to step 7 (see steps 9.1 and 9.2).
- f. The substeps in step 7 are executed. The reseed function will be called; any *additional\_input* provided by the consuming application in the generate request will be used during reseeding. The new values of the internal state are acquired, any *additional\_input* provided by the consuming application in the generate request is replaced by a *Null* string, and the *reseed\_required\_flag* is cleared.
- g. The generate algorithm is called (again) in step 8, the check of the returned *status* is made in step 9, and (presumably) step 10 is then executed.

#### 9.4.3 Handling Prediction Resistance Requests

When pseudorandom bits are requested by a consuming application with prediction resistance, the generate function specified in Section 9.4.1 checks that the instantiation allows prediction resistance requests (see step 5 of the generate process); clears the *reseed\_required\_flag* (even though the flag won't be used in this case); executes the substeps of generate process step 7, resulting in a reseed, a new internal state for the instantiation, and setting the additional input to a *Null* value; obtains pseudorandom bits (see generate process step 8); passes through generate process step 9, since another reseed will not be required; and continues with generate process step 10.

#### 9.5 Removing a DRBG Instantiation

The internal state for an instantiation may need to be "released" by erasing the contents of the internal state. The uninstantiate function:

1. Checks the input parameter for validity.
2. Empties the internal state.

The following or an equivalent process **shall** be used to remove (i.e., unstantiate) a DRBG instantiation:

**Unstantiate\_function** (*state\_handle*):

1. *state\_handle*: A pointer or index that indicates the internal state to be “released”.

**Output to a consuming application after un instantiation:**

1. *status*: The status returned from the function. The status will indicate **SUCCESS** or **ERROR\_FLAG**.

**Information retained within the DRBG mechanism boundary after un instantiation:**

An empty internal state.

**Unstantiate Process:**

1. If *state\_handle* indicates an invalid state, then return an **ERROR\_FLAG**.
2. Erase the contents of the internal state indicated by *state\_handle*.
3. Return **SUCCESS**.

## 10 DRBG Algorithm Specifications

### 10.1 Overview

Several DRBG mechanisms are specified in this Standard. The selection of a DRBG mechanism depends on several factors, including the security strength to be supported and what cryptographic primitives are available. An analysis of the consuming application's requirements for random numbers **should** be conducted in order to select an appropriate DRBG mechanism. A detailed discussion on DRBG mechanism selection is provided in Annex D. Pseudocode examples for each DRBG mechanism are provided in Annex E. Conversion specifications required for the DRBG mechanism implementations (e.g., between integers and bitstrings) are provided in Annex B.

### 10.2 Deterministic RBG Based on Hash Functions

#### 10.2.1 Discussion

A DRBG mechanism is based on a hash function that is non-invertible or one-way. The hash-based DRBG mechanism specified in this Standard has been designed to use any Approved hash function and may be used by consuming applications requiring various security strengths, providing that the appropriate hash function is used and sufficient entropy is obtained for the seed.

The maximum security strength that could be supported by each DRBG based on a hash function is the security strength of the hash function used; see the ASC X9 Registry for hash function usage. This Standard supports only four security strengths for DRBGs: 112, 128, 192, and 256 bits. Table 2 specifies the values that **shall** be used for the function envelopes and DRBG algorithm for each Approved hash function.

**Table 2: Definitions for the Hash-Based DRBG Mechanisms**

	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512
Supported security strengths	See ASC X9 Registry 0003				
<i>highest_supported_security_strength</i>	See ASC X9 Registry 0003				
Output Block Length ( <i>outlen</i> )	160	224	256	384	512
Required minimum entropy for instantiate and reseed	<i>security_strength</i>				
Minimum entropy input length ( <i>min_length</i> )	<i>security_strength</i>				
Maximum entropy input length ( <i>max_length</i> )	$\leq 2^{35}$ bits				
Seed length ( <i>seedlen</i> )	440	440	440	888	888
Maximum personalization string length ( <i>max_personalization_string_length</i> )	$\leq 2^{35}$ bits				
Maximum additional input length ( <i>max_additional_input_length</i> )	$\leq 2^{35}$ bits				
<i>max_number_of_bits_per_request</i>	$\leq 2^{19}$ bits				
Number of requests between reseeds ( <i>reseed_interval</i> )	$\leq 2^{48}$				

Note that since SHA-224 is based on SHA-256, and SHA-384 is based on SHA-512, there is no efficiency benefit for using the SHA-224 or SHA-384.

The value for *seedlen* is determined by subtracting the count field (in the hash function specification) and one byte of padding from the hash function input block length; in the case of SHA-1, SHA-224 and SHA 256,  $seedlen = 512 - 64 - 8 = 440$ ; for SHA-384 and SHA-512,  $seedlen = 1024 - 128 - 8 = 888$ .

## 10.2.2 HMAC\_DRBG

### 10.2.2.1 Discussion

**HMAC\_DRBG** uses multiple occurrences of an Approved keyed hash function, which is based on an Approved hash function. This DRBG mechanism uses the **HMAC\_DRBG\_Update** function specified in Section 10.2.2.2 and the **HMAC** function within the **HMAC\_DRBG\_Update** function as the derivation function during instantiation and reseeding. The same 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.

Figure 7 depicts the **HMAC\_DRBG** in three stages. **HMAC\_DRBG** is specified using an internal function (**HMAC\_DRBG\_Update**). This function is called by the **HMAC\_DRBG** instantiate, generate and reseed algorithms to adjust the internal state when new entropy or additional input is provided, as well as to update the internal state after pseudorandom bits are generated. The operations in the top portion of the figure are only performed if the additional input is not null. Figure 8 depicts the **HMAC\_DRBG\_Update** function.

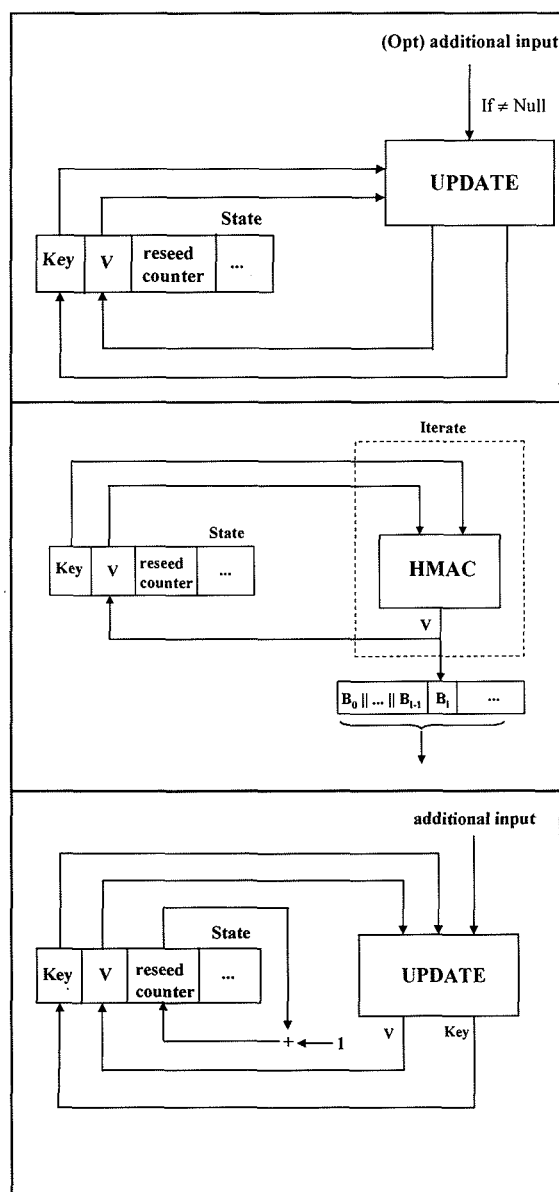


Figure 7: HMAC\_DRBG Generate Function



### 10.2.2.2 Specifications

#### 10.2.2.2.1 HMAC\_DRBG Internal State

The internal state for **HMAC\_DRBG** consists of:

1. The *working\_state*:
  - a. The value  $V$  of *outlen* bits, which is updated each time another *outlen* bits of output are produced (where *outlen* is specified in Table 2 of Section 10.2.1).
  - b. The *outlen*-bit *Key*, which is updated at least once each time that the DRBG mechanism generates pseudorandom bits.
  - c. A counter (*reseed\_counter*) that indicates the number of requests for pseudorandom bits since instantiation or reseeding.
2. Administrative information:
  - a. The *security\_strength* of the DRBG instantiation.
  - b. A *prediction\_resistance\_flag* that indicates whether or not a prediction resistance capability is required for the DRBG instantiation.

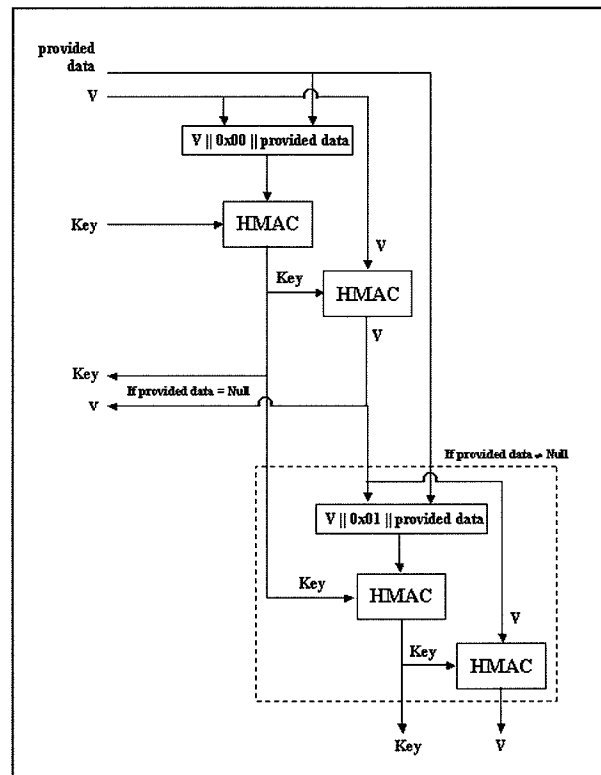


Figure 8: HMAC\_DRBG Update Function

The value of *Key* is the critical values of the internal state upon which the security of this DRBG mechanism depends (i.e., *Key* is the “secret value” of the internal state).

#### 10.2.2.2.2 The Update Function (CTR\_DRBG\_Update)

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 specified in FIPS 198 using the hash function selected for the DRBG mechanism from Table 2 in Section 10.2.1.

The following or an equivalent process **shall** be used as the **HMAC\_DRBG\_Update** function.

**HMAC\_DRBG\_Update** (*provided\_data*,  $K$ ,  $V$ ):

1. *provided\_data*: The data to be used.
2.  $K$ : The current value of *Key*.
3.  $V$ : The current value of  $V$ .

**Output:**

1.  $K$ : The new value for  $Key$ .
2.  $V$ : The new value for  $V$ .

**HMAC\_DRBG Update Process:**

1.  $K = \text{HMAC}(K, V \parallel 0x00 \parallel \text{provided\_data})$ .
2.  $V = \text{HMAC}(K, V)$ .
3. If ( $\text{provided\_data} = \text{Null}$ ), then return  $K$  and  $V$ .
4.  $K = \text{HMAC}(K, V \parallel 0x01 \parallel \text{provided\_data})$ .
5.  $V = \text{HMAC}(K, V)$ .
6. Return  $K$  and  $V$ .

**10.2.2.2.3 Instantiation of HMAC\_DRBG**

Notes for the instantiate function specified in Section 9.2:

The instantiation of **HMAC\_DRBG** requires a call to the instantiate function specified in Section 9.2. Process step 9 of that function calls the instantiate algorithm specified in this section. For this DRBG mechanism, instantiate process step 5 is omitted. The values of *highest\_supported\_security\_strength* and *min\_length* are provided in Table 2 of Section 10.2.1. The contents of the internal state are provided in Section 10.2.2.2.1.

The instantiate algorithm:

Let **HMAC\_DRBG\_Update** be the function specified in Section 10.2.2.2.2. The output block length (*outlen*) is provided in Table 2 of Section 10.2.1.

The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG mechanism (see step 9 of the instantiate process in Section 9.2):

**HMAC\_DRBG\_Instantiate\_algorithm** (*entropy\_input*, *nonce*, *personalization\_string*):

1. *entropy\_input*: The string of bits obtained from the source of entropy input.
2. *nonce*: A string of bits as specified in Section 8.4.2.
3. *personalization\_string*: The personalization string received from the consuming application. Note that the length of the *personalization\_string* may be zero.

**Output:**

1. *initial\_working\_state*: The initial values for  $V$ ,  $Key$  and *reseed\_counter* (see Section 10.2.2.2.1).

**HMAC\_DRBG Instantiate Process:**

1.  $\text{seed\_material} = \text{entropy\_input} \parallel \text{nonce} \parallel \text{personalization\_string}$ .
  2.  $Key = 0x00\ 00\dots00$ . Comment: *outlen* bits.
  3.  $V = 0x01\ 01\dots01$ . Comment: *outlen* bits.
- Comment: Update  $Key$  and  $V$ .

4.  $(Key, V) = \text{HMAC\_DRBG\_Update}(seed\_material, Key, V)$ .
5.  $reseed\_counter = 1$ .
6. Return  $V$ ,  $Key$  and  $reseed\_counter$  as the *initial\_working\_state*.

#### 10.2.2.2.4 Reseeding an HMAC\_DRBG Instantiation

Notes for the reseed function specified in Section 9.3:

The reseeding of an **HMAC\_DRBG** instantiation requires a call to the reseed function specified in Section 9.3. Process step 5 of that function calls the reseed algorithm specified in this section. The values for *min\_length* are provided in Table 2 of Section 10.2.1.

The reseed algorithm:

Let **HMAC\_DRBG\_Update** be the function specified in Section 10.2.2.2.2. The following process or its equivalent **shall** be used as the reseed algorithm for this DRBG mechanism (see step 5 of the reseed process in Section 9.3):

**HMAC\_DRBG\_Reseed\_algorithm** (*working\_state*, *entropy\_input*, *additional\_input*):

1. *working\_state*: The current values for  $V$ ,  $Key$  and  $reseed\_counter$  (see Section 10.2.2.2.1).
2. *entropy\_input*: The string of bits obtained from the source of entropy input.
3. *additional\_input*: The additional input string received from the consuming application. If the input of *additional\_input* is not supported by the implementation, then process step 1 of the **HMAC\_DRBG** reseed process is modified to remove the *additional\_input*.

**Output:**

1. *new\_working\_state*: The new values for  $V$ ,  $Key$  and  $reseed\_counter$ .

**HMAC\_DRBG Reseed Process:**

1.  $seed\_material = entropy\_input \parallel additional\_input$ .
2.  $(Key, V) = \text{HMAC\_DRBG\_Update}(seed\_material, Key, V)$ .
3.  $reseed\_counter = 1$ .
4. Return  $V$ ,  $Key$  and  $reseed\_counter$  as the *new\_working\_state*.

#### 10.2.2.2.5 Generating Pseudorandom Bits Using HMAC\_DRBG

Notes for the generate function specified in Section 9.4:

The generation of pseudorandom bits using an **HMAC\_DRBG** instantiation requires a call to the generate function specified in Section 9.4. Process step 8 of that function calls the generate algorithm specified in this section. The values for *outlen* and *max\_number\_of\_bits\_per\_request* are provided in Table 2 of Section 10.2.1.

The generate algorithm :

Let **HMAC** be the keyed hash function specified in ASC X9 Registry 00004 using the hash function selected for the DRBG mechanism. The value for *reseed\_interval* is defined in Table 2 of Section 10.2.1.

The following process or its equivalent **shall** be used as the generate algorithm for this DRBG mechanism (see step 8 of the generate process in Section 9.4):

**HMAC\_DRBG\_Generate\_algorithm** (*working\_state*, *requested\_number\_of\_bits*, *additional\_input*):

1. *working\_state*: The current values for *V*, *Key* and *reseed\_counter* (see Section 10.2.2.2.1).
2. *requested\_number\_of\_bits*: The number of pseudorandom bits to be returned to the generate function.
3. *additional\_input*: The additional input string received from the consuming application. If the input of *additional\_input* is not supported by an implementation, then step 2 of the **HMAC\_DRBG** generate process is omitted. If the implementation allows *additional\_input*, but a given request does not provide any *additional\_input*, or *additional\_input* is not supported, then a *Null* string **shall** be used as the *additional\_input* in step 6 of the **HMAC\_DRBG** generate process.

**Output:**

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS** or indicate that a reseed is required before the requested pseudorandom bits can be generated.
2. *returned\_bits*: The pseudorandom bits to be returned to the generate function.
3. *new\_working\_state*: The new values for *V*, *Key* and *reseed\_counter*.

**HMAC\_DRBG Generate Process:**

1. If *reseed\_counter* > *reseed\_interval*, then return an indication that a reseed is required.
2. If *additional\_input* ≠ *Null*, then (*Key*, *V*) = **HMAC\_DRBG\_Update**(*additional\_input*, *Key*, *V*).
3. *temp* = *Null*.
4. While (**len** (*temp*) < *requested\_number\_of\_bits*) do:
  - 4.1 *V* = **HMAC** (*Key*, *V*).
  - 4.2 *temp* = *temp* || *V*.
5. *returned\_bits* = Leftmost *requested\_number\_of\_bits* of *temp*.
6. (*Key*, *V*) = **HMAC\_DRBG\_Update** (*additional\_input*, *Key*, *V*).
7. *reseed\_counter* = *reseed\_counter* + 1.
8. Return **SUCCESS**, *returned\_bits*, and the new values of *Key*, *V* and *reseed\_counter* as the *new\_working\_state*.

### 10.3 DRBG Mechanisms Based on Block Ciphers

#### 10.3.1 Discussion

A block cipher DRBG is based on a block cipher algorithm. The block cipher DRBG mechanism specified in this Standard has been designed to use any Approved block cipher algorithm (see the ASC X9 Registry) and may be used by consuming applications requiring various security strengths, providing that the appropriate block cipher algorithm and key length are used, and sufficient entropy is obtained for the seed.

The maximum security strength that can be supported by each DRBG based on a block cipher is the security strength of the block cipher and key size used; see the ASC X9 Registry for guidance.

#### 10.3.2 CTR\_DRBG

##### 10.3.2.1 CTR\_DRBG Description

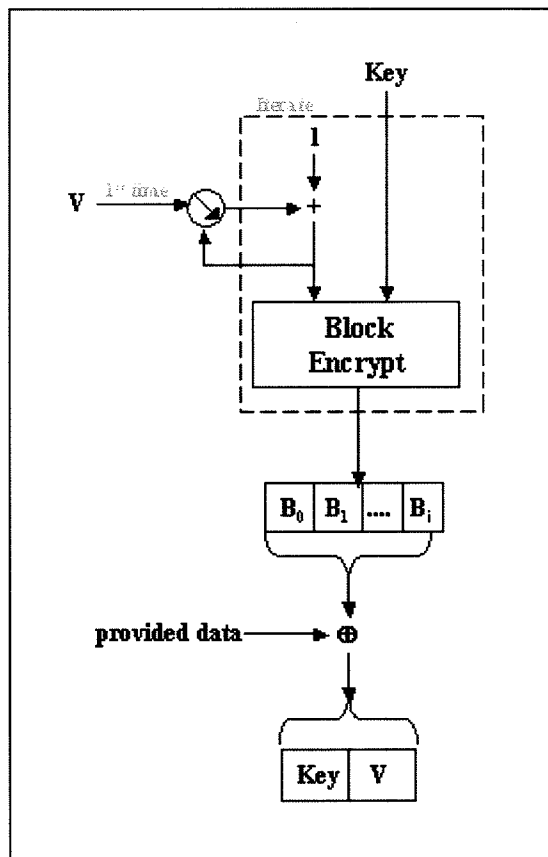
**CTR\_DRBG** uses an Approved block cipher algorithm in the counter mode (see ASC Registry 00002). The same block cipher algorithm and key length **shall** be used for all block cipher operations. The block cipher algorithm and key length **shall** meet or exceed the security requirements of the consuming application.

**CTR\_DRBG** is specified using an internal function (**CTR\_DRBG\_Update**). Figure 9 depicts the **CTR\_DRBG\_Update** function. This function is called by the instantiate, generate and reseed algorithms to adjust the internal state when new entropy or additional input is provided, as well as to update the internal state after pseudorandom bits are generated. Figure 10 depicts the **CTR\_DRBG** in three stages. The operations in the top portion of the figure are only performed if the additional input is not null.

Table 3 specifies the values that **shall** be used for the function envelopes and DRBG algorithms.

**Table 3: Definitions for the CTR\_DRBG**

	3 Key TDEA	AES-128	AES-192	AES-256
<b>Supported security strengths</b>	See ASC X9 Registry			
<i>highest_supported_security_strength</i>	See ASC X9 Registry			
<b>Output block length (<i>outlen</i>)</b>	64	128	128	128



**Figure 9: CTR\_DRBG Update Function**

	3 Key TDEA	AES-128	AES-192	AES-256
Key length ( <i>keylen</i> )	168	128	192	256
Required minimum entropy for instantiate and reseed	<i>security_strength</i>			
Seed length ( <i>seedlen</i> = <i>outlen</i> + <i>keylen</i> )	232	256	320	384
If a derivation function is used:				
a. Minimum entropy input length ( <i>min_length</i> )	<i>security_strength</i>			
b. Maximum entropy input length ( <i>max_length</i> )	$\leq 2^{35}$ bits			
c. Maximum personalization string length ( <i>max_personalization_string_length</i> )	$\leq 2^{35}$ bits			
d. Maximum additional_input length ( <i>max_additional_input_length</i> )	$\leq 2^{35}$ bits			
If a derivation function is not used:				
a. Minimum entropy input length ( <i>min_length</i> = <i>outlen</i> + <i>keylen</i> )	<i>seedlen</i>			
b. Maximum entropy input length ( <i>max_length</i> ) ( <i>outlen</i> + <i>keylen</i> )	<i>seedlen</i>			
c. Maximum personalization string length ( <i>max_personalization_string_length</i> )	<i>seedlen</i>			
d. Maximum additional_input length ( <i>max_additional_input_length</i> )	<i>seedlen</i>			
<i>max_number_of_bits_per_request</i>	$\leq 2^{13}$	$\leq 2^{19}$		
Number of requests between reseeds ( <i>reseed_interval</i> )	$\leq 2^{32}$	$\leq 2^{48}$		

The **CTR\_DRBG** may be implemented to use the block cipher derivation function specified in Section 10.5.3 during instantiation and reseeding. However, the DRBG imechanism is specified to allow an implementation tradeoff with respect to the use of this derivation function. The use of the derivation function is optional if either of the following is available to provide entropy input when requested:

- An Approved RBG with a security strength equal to or greater than the required security strength of the **CTR\_DRBG** instantiation, or
- An Approved conditioned entropy source.

Otherwise, the derivation function **shall** be used. Table 3 provides lengths required for the *entropy\_input*, *personalization\_string* and *additional\_input* for each case.

When a derivation function is not used by an implementation, the seed construction **shall not** use a nonce<sup>2</sup> (see Section 8.4.2).

When using TDEA as the selected block cipher algorithm, the keys **shall** be handled as 64-bit blocks containing 56 bits of key and 8 bits of parity as specified for the TDEA engine in ANS X9.52.

### 10.3.2.2 Specifications

#### 10.3.2.2.1 CTR\_DRBG Internal State

The internal state for **CTR\_DRBG** consists of:

1. The *working\_state*:
  - a. The value *V* of *outlen* bits, which is updated each time another *outlen* bits of output are produced (see Table 3 in Section 10.3.2.1).
  - b. The *keylen*-bit *Key*, which is updated whenever a predetermined number of output blocks are generated.
  - c. A counter (*reseed\_counter*) that indicates the number of requests for pseudorandom bits since instantiation or reseeding.
2. Administrative information:

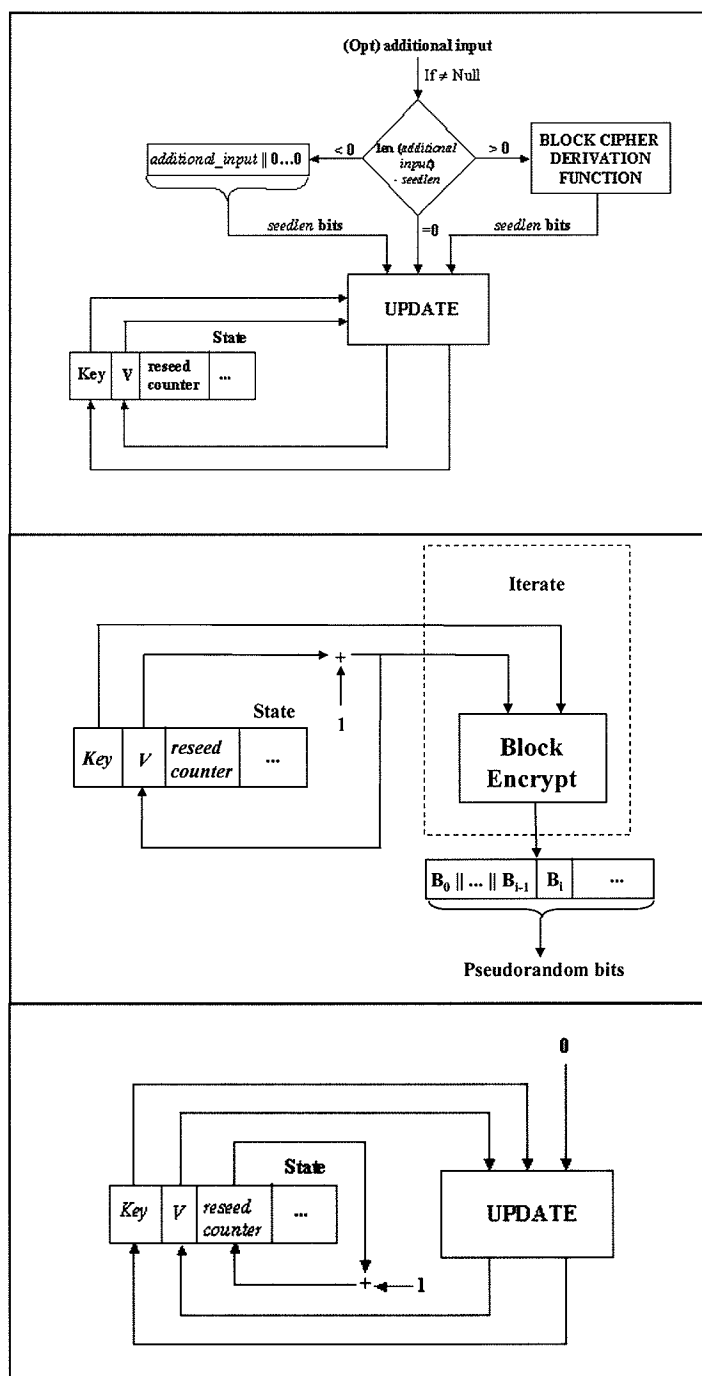


Figure 10: CTR-DRBG

<sup>2</sup> The specifications in this Standard do not accommodate the special treatment required for a nonce in this case.

- a. The *security\_strength* of the DRBG instantiation.
- b. A *prediction\_resistance\_flag* that indicates whether or not a prediction resistance capability is required for the DRBG instantiation.

The values of  $V$  and  $Key$  are the critical values of the internal state upon which the security of this DRBG mechanism depends (i.e.,  $V$  and  $Key$  are the "secret values" of the internal state).

#### 10.3.2.2.2 The Update Function (CTR\_DRBG\_Update)

The **CTR\_DRBG\_Update** function updates the internal state of the **CTR\_DRBG** using the *provided\_data*. The values for *outlen*, *keylen* and *seedlen* are provided in Table 3 of Section 10.3.2.1. The block cipher operation in step 2.2 of the **CTR\_DRBG\_Update** process uses the selected block cipher algorithm (also see Section 10.5.4).

The following or an equivalent process **shall** be used as the **CTR\_DRBG\_Update** function:

##### **CTR\_DRBG\_Update** (*provided\_data*, $Key$ , $V$ ):

1. *provided\_data*: The data to be used. This must be exactly *seedlen* bits in length; this length is guaranteed by the construction of the *provided\_data* in the instantiate, reseed and generate functions.
2.  $Key$ : The current value of  $Key$ .
3.  $V$ : The current value of  $V$ .

##### **Output:**

1.  $K$ : The new value for  $Key$ .
2.  $V$ : The new value for  $V$ .

##### **CTR\_DRBG Update Process:**

1.  $temp = Null$ .
2. While (**len** ( $temp$ ) < *seedlen*) do
  - 2.1  $V = (V + 1) \bmod 2^{outlen}$ .
  - 2.2  $output\_block = \mathbf{Block\_Encrypt}(Key, V)$ .
  - 2.3  $temp = temp \parallel output\_block$ .
3.  $temp$  = Leftmost *seedlen* bits of  $temp$ .
4.  $temp = temp \oplus provided\_data$ .
5.  $Key$  = Leftmost *keylen* bits of  $temp$ .
6.  $V$  = Rightmost *outlen* bits of  $temp$ .
7. Return the new values of  $Key$  and  $V$ .

#### 10.3.2.2.3 Instantiation of CTR\_DRBG

Notes for the instantiate function specified in Section 9.2:



The instantiation of **CTR\_DRBG** requires a call to the instantiate function specified in Section 9.2. Process step 9 of that function calls the instantiate algorithm specified in this section. For this DRBG mechanism, step 5 of the instantiate function is omitted. The values of *highest\_supported\_security\_strength* and *min\_length* are provided in Table 3 of Section 10.3.2.1. The contents of the internal state are provided in Section 10.3.2.2.1.

The instantiate algorithm:

For this DRBG mechanism, there are two cases for the processing. In each case, let **CTR\_DRBG\_Update** be the function specified in Section 10.3.2.2.2. The output block length (*outlen*), key length (*keylen*), seed length (*seedlen*) and *security\_strengths* for the block cipher algorithms are provided in Table 3 of Section 10.3.2.1.

#### 10.3.2.2.3.1 The Process Steps for Instantiation When Full Entropy is Available for the Entropy Input, and a Derivation Function is Not Used

The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG mechanism:

**CTR\_DRBG\_Instantiate\_algorithm** (*entropy\_input*, *personalization\_string*):

1. *entropy\_input*: The string of bits obtained from the source of entropy input.
2. *personalization\_string*: The personalization string received from the consuming application. Note that the length of the *personalization\_string* may be zero.

**Output:**

1. *initial\_working\_state*: The initial values for *V*, *Key*, and *reseed\_counter* (see Section 10.3.2.2.1).

**CTR\_DRBG Instantiate Process:**

1. *temp* = **len** (*personalization\_string*).

Comment: Ensure that the length of the *personalization\_string* is exactly *seedlen* bits. The maximum length was checked in Section 9.2, processing step 3, using Table 3 to define the maximum length.

2. If (*temp* < *seedlen*), then *personalization\_string* = *personalization\_string* ||  $0^{seedlen - temp}$ .

3. *seed\_material* = *entropy\_input*  $\oplus$  *personalization\_string*.

4. *Key* =  $0^{keylen}$ .

Comment: *keylen* bits of zeros.

5. *V* =  $0^{outlen}$ .

Comment: *outlen* bits of zeros.

6. (*Key*, *V*) = **CTR\_DRBG\_Update** (*seed\_material*, *Key*, *V*).

7. *reseed\_counter* = 1.

8. **Return** *V*, *Key*, and *reseed\_counter* as the *initial\_working\_state*.

#### 10.3.2.2.3.2 The Process Steps for Instantiation When a Derivation Function is Used

Let **Block\_Cipher\_df** be the derivation function specified in Section 10.5.3 using the chosen block cipher algorithm and key size.

The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG mechanism:

**CTR\_DRBG\_Instantiate\_algorithm** (*entropy\_input*, *nonce*, *personalization\_string*):

1. *entropy\_input*: The string of bits obtained from the source of entropy input.
2. *nonce*: A string of bits as specified in Section 8.4.2.
3. *personalization\_string*: The personalization string received from the consuming application. Note that the length of the *personalization\_string* may be zero.

**Output:**

1. *initial\_working\_state*: The initial values for *V*, *Key*, and *reseed\_counter* (see Section 10.3.2.2.1).

**CTR\_DRBG\_Instantiate\_Process:**

1.  $seed\_material = entropy\_input \parallel nonce \parallel personalization\_string$ .  
 Comment: Ensure that the length of the *seed\_material* is exactly *seedlen* bits.
2.  $seed\_material = \mathbf{Block\_Cipher\_df}(seed\_material, seedlen)$ .
3.  $Key = 0^{keylen}$ .                      Comment: *keylen* bits of zeros.
4.  $V = 0^{outlen}$ .                      Comment: *outlen* bits of zeros.
5.  $(Key, V) = \mathbf{CTR\_DRBG\_Update}(seed\_material, Key, V)$ .
6.  $reseed\_counter = 1$ .
7. **Return** *V*, *Key*, and *reseed\_counter* as the *initial\_working\_state*.

#### 10.3.2.2.4 Reseeding a CTR\_DRBG Instantiation

Notes for the reseed function specified in Section 9.3:

The reseed of a **CTR\_DRBG** instantiation requires a call to the reseed function specified in Section 9.3. Process step 5 of that function calls the reseed algorithm specified in this section. The values for *min\_length* are provided in Table 3 of Section 10.3.2.1.

The reseed algorithm:

For this DRBG mechanism, there are two cases for the processing. In each case, let **CTR\_DRBG\_Update** be the function specified in Section 10.3.2.2.2. The seed length (*seedlen*) is provided in Table 3 of Section 10.3.2.1.

#### 10.3.2.2.4.1 The Process Steps for Reseeding When Full Entropy is Available for the Entropy Input, and a Derivation Function is Not Used

The following process or its equivalent **shall** be used as the reseed algorithm for this DRBG mechanism (see step 5 of the reseed process in Section 9.3):

**CTR\_DRBG\_Reseed\_algorithm** (*working\_state*, *entropy\_input*, *additional\_input*):

1. *working\_state*: The current values for *V*, *Key*, *previous\_output\_block* and *reseed\_counter* (see Section 10.3.2.2.1).
2. *entropy\_input*: The string of bits obtained from the source of entropy input.
3. *additional\_input*: The additional input string received from the consuming application. If the input of *additional\_input* is not supported by an implementation, then reseed process steps 1 to 3 below are replaced by:

*seed\_material* = *entropy\_input*.

That is, steps 1-3 collapse into the above step.

#### Output :

1. *new\_working\_state*: The new values for *V*, *Key*, and *reseed\_counter*.

#### CTR\_DRBG Reseed Process

1. *temp* = **len** (*additional\_input*).

Comment: Ensure that the length of the *additional\_input* is exactly *seedlen* bits. The maximum length was checked in Section 9.3, processing step 2, using Table 3 to define the maximum length.

2. If (*temp* < *seedlen*), then *additional\_input* = *additional\_input* || 0<sup>*seedlen* - *temp*</sup>.
3. *seed\_material* = *entropy\_input* ⊕ *additional\_input*.
4. (*Key*, *V*) = **CTR\_DRBG\_Update** (*seed\_material*, *Key*, *V*).
5. *reseed\_counter* = 1.
6. **Return** *V*, *Key* and *reseed\_counter* as the *new\_working\_state*.

#### 10.3.2.2.4.2 The Process Steps for Reseeding When a Derivation Function is Used

Let **Block\_Cipher\_df** be the derivation function specified in Section 10.5.3 using the chosen block cipher algorithm and key size.

The following process or its equivalent **shall** be used as the reseed algorithm for this DRBG mechanism (see reseed process step 5 of Section 9.3):

**CTR\_DRBG\_Reseed\_algorithm** (*working\_state*, *entropy\_input*, *additional\_input*):

1. *working\_state*: The current values for *V*, *Key*, *previous\_output\_block* and *reseed\_counter* (see Section 10.3.2.2.1).

2. *entropy\_input*: The string of bits obtained from the source of entropy input.
3. *additional\_input*: The additional input string received from the consuming application. If the input of *additional\_input* is not supported by an implementation, then reseed process steps 1 to 3 below are replaced by:

*seed\_material* = *entropy\_input*.

That is, steps 1-3 collapse into the above step.

#### Output :

1. *new\_working\_state*: The new values for *V*, *Key*, and *reseed\_counter*.

#### CTR\_DRBG Reseed Process:

1. *seed\_material* = *entropy\_input* || *additional\_input*.

Comment: Ensure that the length of the *seed\_material* is exactly *seedlen* bits.

2. *seed\_material* = **Block\_Cipher\_df**(*seed\_material*, *seedlen*).
3. (*Key*, *V*) = **CTR\_DRBG\_Update**(*seed\_material*, *Key*, *V*).
4. *reseed\_counter* = 1.
5. **Return** *V*, *Key*, and *reseed\_counter* as the *new\_working\_state*.

#### 10.3.2.2.5 Generating Pseudorandom Bits Using CTR\_DRBG

Notes for the generate function specified in Section 9.4:

The generation of pseudorandom bits using a **CTR\_DRBG** instantiation requires a call to the generate function specified in Section 9.4. Process step 8 of that function calls the generate algorithm specified in this section. The values for *max\_number\_of\_bits\_per\_request*, *max\_additional\_input\_length*, and *outlen* are provided in Table 3 of Section 10.3.2.1. If the derivation function is not used, then the maximum allowed length of *additional\_input* = *seedlen*.

For this DRBG mechanism, there are two cases for the processing. For each case, let **CTR\_DRBG\_Update** be the function specified in Section 10.3.2.2.2, and let **Block\_Encrypt** be the function specified in Section 10.5.4. The seed length (*seedlen*) and the value of *reseed\_interval* are provided in Table 3 of Section 10.3.2.1.

##### 10.3.2.2.5.1 The Process Steps for Generating Pseudorandom Bits When a Derivation Function is Not Used for the DRBG Implementation

The following process or its equivalent **shall** be used as the generate algorithm for this DRBG mechanism (see step 8 of the generate process in Section 9.4.1):

**CTR\_DRBG\_Generate\_algorithm**(*working\_state*, *requested\_number\_of\_bits*, *additional\_input*):

1. *working\_state*: The current values for *V*, *Key*, and *reseed\_counter* (see Section 10.3.2.2.1).

2. *requested\_number\_of\_bits*: The number of pseudorandom bits to be returned to the generate function.
3. *additional\_input*: The additional input string received from the consuming application. If *additional\_input* will never be allowed, then step 2 becomes:

$$\text{additional\_input} = 0^{\text{seedlen}}.$$

#### Output:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS**, or indicate that a reseed is required before the requested pseudorandom bits can be generated.
2. *returned\_bits*: The pseudorandom bits returned to the generate function.
3. *working\_state*: The new values for *V*, *Key*, and *reseed\_counter*.

#### CTR\_DRBG Generate Process:

1. If *reseed\_counter* > *reseed\_interval*, then return an indication that a reseed is required.
2. If (*additional\_input* ≠ Null), then

Comment: Ensure that the length of the *additional\_input* is exactly *seedlen* bits. The maximum length was checked in Section 9.4.1, processing step 4, using Table 3 to define the maximum length. If the length of the *additional\_input* is < *seedlen*, pad with zero bits.

2.1 *temp* = **len** (*additional\_input*).

2.2 If (*temp* < *seedlen*), then  
       *additional\_input* = *additional\_input* ||  $0^{\text{seedlen} - \text{temp}}$ .

2.3 (*Key*, *V*) = **CTR\_DRBG\_Update** (*additional\_input*, *Key*, *V*).

Else *additional\_input* =  $0^{\text{seedlen}}$ .

3. *temp* = Null.
4. While (**len** (*temp*) < *requested\_number\_of\_bits*) do:
  - 4.1  $V = (V + 1) \bmod 2^{\text{outlen}}$ .
  - 4.2 *output\_block* = **Block\_Encrypt** (*Key*, *V*).
  - 4.3 *temp* = *temp* || *output\_block*.
5. *returned\_bits* = Leftmost *requested\_number\_of\_bits* of *temp*.

Comment: Update for backtracking resistance.

6.  $(Key, V) = \text{CTR\_DRBG\_Update}(\text{additional\_input}, Key, V)$ .
7.  $\text{reseed\_counter} = \text{reseed\_counter} + 1$ .
8. Return **SUCCESS** and *returned\_bits*; also return *Key*, *V*, and *reseed\_counter* as the *new\_working\_state*.

#### 10.3.2.2.5.2 The Process Steps for Generating Pseudorandom Bits When a Derivation Function is Used for the DRBG Implementation

The **Block\_Cipher\_df** is specified in Section 10.5.3 and **shall** be implemented using the chosen block cipher algorithm and key size.

The following process or its equivalent **shall** be used as generate algorithm for this DRBG mechanism (see step 8 of the generate process in Section 9.4.1):

**CTR\_DRBG\_Generate\_algorithm** (*working\_state*, *requested\_number\_of\_bits*, *additional\_input*):

1. *working\_state*: The current values for *V*, *Key*, and *reseed\_counter* (see Section 10.3.2.2.1).
2. *requested\_number\_of\_bits*: The number of pseudorandom bits to be returned to the generate function.
3. *additional\_input*: The additional input string received from the consuming application. If *additional\_input* will never be allowed, then step 2 becomes:

$$\text{additional\_input} = 0^{\text{seedlen}}.$$

#### Output:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS**, or indicate that a reseed is required before the requested pseudorandom bits can be generated.
2. *returned\_bits*: The pseudorandom bits returned to the generate function.
3. *working\_state*: The new values for *V*, *Key*, and *reseed\_counter*.

#### CTR\_DRBG Generate Process:

1. If  $\text{reseed\_counter} > \text{reseed\_interval}$ , then return an indication that a reseed is required.
2. If (*additional\_input*  $\neq$  Null), then
  - 2.1  $\text{additional\_input} = \text{Block\_Cipher\_df}(\text{additional\_input}, \text{seedlen})$ .
  - 2.2  $(Key, V) = \text{CTR\_DRBG\_Update}(\text{additional\_input}, Key, V)$ .
 Else  $\text{additional\_input} = 0^{\text{seedlen}}$ .
3. *temp* = Null.
4. While (**len** (*temp*) < *requested\_number\_of\_bits*) do:
  - 4.1  $V = (V + 1) \bmod 2^{\text{outlen}}$ .

- 4.2  $output\_block = \mathbf{Block\_Encrypt}(Key, V)$ .
- 4.3  $temp = temp \parallel output\_block$ .
5.  $returned\_bits = \text{Leftmost } requested\_number\_of\_bits \text{ of } temp$ .  
Comment: Update for backtracking resistance.
6.  $(Key, V) = \mathbf{CTR\_DRBG\_Update}(additional\_input, Key, V)$ .
7.  $reseed\_counter = reseed\_counter + 1$ .
8. Return **SUCCESS** and  $returned\_bits$ ; also return  $Key$ ,  $V$ , and  $reseed\_counter$  as the  $new\_working\_state$ .

## 10.4 DRBG Mechanisms Based on Number Theoretic Problems

### 10.4.1 Discussion

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. Section 10.4.2 specifies a DRBG mechanism based on the elliptic curve discrete logarithm problem.

### 10.4.2 Dual Elliptic Curve Deterministic RBG (Dual\_EC\_DRBG)

#### 10.4.2.1 Discussion

The **Dual\_EC\_DRBG** is based on the following hard problem, sometimes known as the “elliptic curve discrete logarithm problem” (ECDLP): given points  $P$  and  $Q$  on an elliptic curve of order  $n$ , find  $a$  such that  $Q = aP$ .

**Dual\_EC\_DRBG** uses an initial seed that is  $2 * security\_strength$  bits in length to initiate the generation of  $outlen$ -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. For all the NIST curves given in this Standard for the DRBG,  $m$  is at least twice the *security\_strength*, and never less than 256. Throughout this DRBG mechanism specification,  $m$  will be referred to as *seedlen*; the term “*seedlen*” is appropriate because the internal state of **Dual\_EC\_DRBG** is used as a “seed” for the random block it produces. Figure 11 depicts the **Dual\_EC\_DRBG**.

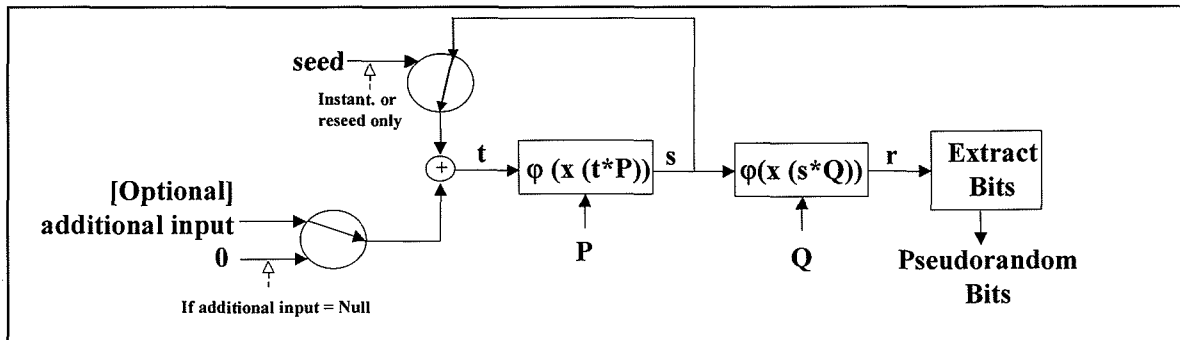


Figure 11: Dual\_EC\_DRBG

The instantiation of this DRBG mechanism requires the selection of an appropriate elliptic curve and curve points specified in Annex A.1 for the desired security strength. The *seed* used to determine the initial value ( $s$ ) of the DRBG mechanism **shall** have entropy that is at least *security\_strength* bits. Further requirements for the *seed* are provided in Section 8.2. This DRBG mechanism uses the derivation function specified in Section 10.5.2 during instantiation and reseeding.

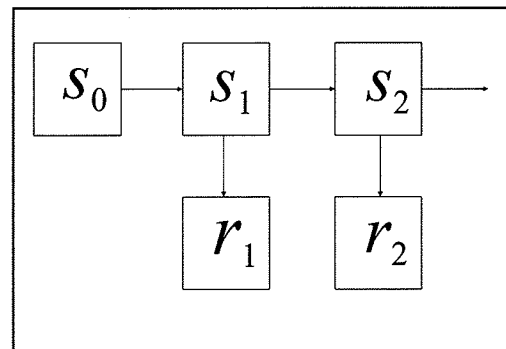
Backtracking resistance is inherent in the algorithm, even if the internal state is compromised. As shown in Figure 12, **Dual\_EC\_DRBG** generates a *seedlen*-bit number for each step  $i = 1, 2, 3, \dots$ , as follows:



$$s_i = \varphi(x(s_{i-1} * P))$$

$$r_i = \varphi(x(s_i * Q)).$$

Each arrow in the figure represents an Elliptic Curve scalar multiplication operation, followed by the extraction of the  $x$  coordinate for the resulting point and for the random output  $r_i$  followed by truncation to produce the output (formal definitions for  $\varphi$  and  $x$  are given in Section 10.4.2.2.4). Following a line in the direction of the arrow is the normal operation; inverting the direction implies the ability to solve the ECDLP for that specific curve. An adversary's ability to invert an arrow in the figure implies that the adversary has solved the ECDLP for that specific elliptic curve. Backtracking resistance is built into the design, as knowledge of  $s_1$  does not allow an adversary to determine  $s_0$  (and so forth) unless the adversary is able to solve the ECDLP for that specific curve. In addition, knowledge of  $r_1$  does not allow an adversary to determine  $s_1$  (and so forth) unless the adversary is able to solve the ECDLP for that specific curve.



**Figure 12: Dual\_EC\_DRBG  
Backtracking Resistance**

Table 4 specifies the values that **shall** be used for the envelope and algorithm for each curve. Complete specifications for each curve are provided in Annex A.1. Note that all curves can be instantiated at a security strength lower than the curve's highest possible security strength. For example, the highest security strength that can be supported by curve P-384 is 192 bits; however, this curve can alternatively be instantiated to support only the 112 or 128-bit security strengths).

**Table 4: Definitions for the Dual\_EC\_DRBG**

	P-256	P-384	P-521
<b>Supported security strengths</b>	See the ASC X9 Registry		
<b>Size of the base field (in bits), references throughout as <i>seedlen</i></b>	256	384	521
<b><i>highest_supported_ security_strength</i></b>	See the ASC X9 registry		
<b>Output block length (<i>max_outlen</i> = largest multiple of 8 less than (size of the base field) - (13 + log<sub>2</sub> (the cofactor))</b>	240	368	504
<b>Required minimum entropy for instantiate and reseed</b>	<i>security_strength</i>		
<b>Minimum entropy input length (<i>min_length</i>)</b>	<i>security_strength</i>		
<b>Maximum entropy input length (<i>max_length</i>)</b>	$\leq 2^{13}$ bits		
<b>Maximum personalization string</b>	$\leq 2^{13}$ bits		

	P-256	P-384	P-521
<b>length</b> ( <i>max_personalization_string_length</i> )			
<b>Maximum additional input length</b> ( <i>max_additional_input_length</i> )	$\leq 2^{13}$ bits		
<b>Length of the initial seed</b>	$2 \times \text{security\_strength}$		
<b>Appropriate hash functions</b>	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512	SHA-224, SHA-256, SHA-384, SHA-512	SHA-256, SHA-384, SHA-512
<i>max_number_of_bits_per_request</i>	$\text{max\_outlen} \times \text{reseed\_interval}$		
<b>Number of blocks between reseeding</b> ( <i>reseed_interval</i> )	$\leq 2^{32}$ blocks		

#### 10.4.2.2 Specifications

##### 10.4.2.2.1 Dual\_EC\_DRBG Internal State

The internal state for **Dual\_EC\_DRBG** consists of:

1. The *working\_state*:
  - a. A value ( $s$ ) that determines the current position on the curve.
  - b. The elliptic curve domain parameters ( $\text{seedlen}$ ,  $p$ ,  $a$ ,  $b$ ,  $n$ ), where  $\text{seedlen}$  is the length of the seed;  $p$  is the prime that defines the base field  $F_p$ ;  $a$  and  $b$  are two field elements that define the equation of the curve; and  $n$  is the order of the point  $G$ . If only one curve will be used by an implementation, these parameters need not be present in the *working\_state*.
  - c. Two points  $P$  and  $Q$  on the curve (see Annex A.1); the generating point  $G$  specified in Annex A.1 for the chosen curve will be used as  $P$ . If only one curve will be used by an implementation, these points need not be present in the *working\_state*.
  - d. A counter (*reseed\_counter*) that indicates the number of blocks of random produced by the **Dual\_EC\_DRBG** since the initial seeding or the previous reseeding.
2. Administrative information:
  - a. The *security\_strength* provided by the instance of the DRBG instantiation,
  - b. A *prediction\_resistance\_flag* that indicates whether prediction resistance is required by the DRBG instantiation.

The value of  $s$  is the critical value of the internal state upon which the security of this DRBG mechanism depends (i.e.,  $s$  is the “secret value” of the internal state).

#### 10.4.2.2.2 Instantiation of Dual\_EC\_DRBG

Notes for the instantiate function specified in Section 9.2:

The instantiation of **Dual\_EC\_DRBG** requires a call to the instantiate function specified in Section 9.2. Process step 9 of that function calls the instantiate algorithm in this section.

In process step 5 of the instantiate function, the following step **shall** be performed to select an appropriate curve if multiple curves are available.

5. Using the *security\_strength* and Table 4 in Section 10.4.2.1, select the smallest available curve that has a security strength  $\geq$  *security\_strength*.

The values for *seedlen*, *p*, *a*, *b*, *n*, *P*, *Q* are determined by that curve.

It is recommended that the default values be used for *P* and *Q* as given in Annex A.1. However, an implementation may use different pairs of points, provided that they are verifiably random, as evidenced by the use of the procedure specified in Annex A.2.1 and the self-test procedure described in Annex A.2.2.

The values for *highest\_supported\_security\_strength* and *min\_length* are determined by the selected curve (see Table 4 in Section 10.4.2.1).

The instantiate algorithm :

Let **Hash\_df** be the hash derivation function specified in Section 10.5.2 using an appropriate hash function from Table 4 in Section 10.4.2.1. Let *seedlen* be the appropriate value from Table 4.

The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG mechanism (see step 9 of the instantiate process in Section 9.2):

**Dual\_EC\_DRBG\_Instantiate\_algorithm** (*entropy\_input*, *nonce*, *personalization\_string*):

1. *entropy\_input*: The string of bits obtained from the source of entropy input.
2. *nonce*: A string of bits as specified in Section 8.4.2.
3. *personalization\_string*: The personalization string received from the consuming application. Note that the length of the *personalization\_string* may be zero.

**Output:**

1. *s*: The initial secret value for the *initial\_working\_state*.
2. *reseed\_counter*: The initialized block counter for reseeding.

**Dual\_EC\_DRBG Instantiate Process:**

1.  $seed\_material = entropy\_input \parallel nonce \parallel personalization\_string$ .

Comment: Use a hash function to ensure that the entropy is distributed throughout the bits, and *s* is *m* (i.e., *seedlen*) bits in length.

2.  $s = \text{Hash\_df}(seed\_material, seedlen)$ .
3.  $reseed\_counter = 0$ .

4. Return  $s$ , and  $reseed\_counter$  for the *initial\_working\_state*.

#### 10.4.2.2.3 Reseeding of a Dual\_EC\_DRBG Instantiation

Notes for the reseed function specified in Section 9.3:

The reseed of **Dual\_EC\_DRBG** requires a call to the reseed function specified in Section 9.3. Process step 5 of that function calls the reseed algorithm in this section. The values for *min\_length* are provided in Table 4 of Section 10.4.2.1.

The reseed algorithm :

Let **Hash\_df** be the hash derivation function specified in Section 10.5.2 using an appropriate hash function from Table 4 in Section 10.4.2.1.

The following process or its equivalent **shall** be used to reseed the **Dual\_EC\_DRBG** process after it has been instantiated (see step 5 of the reseed process in Section 9.3):

**Dual\_EC\_DRBG\_Reseed\_algorithm** ( $s$ , *entropy\_input*, *additional\_input*):

1.  $s$ : The current value of the secret parameter in the *working\_state*.
2. *entropy\_input*: The string of bits obtained from the source of entropy input.
3. *additional\_input*: The additional input string received from the consuming application. If the input of a *additional\_input* is not supported by an implementation, then the *additional\_input* term is removed from step 1 of the reseed process, so that step 1 becomes:

$$seed\_material = \text{pad8}(s) \parallel \text{entropy\_input}.$$

**Output:**

1.  $s$ : The new value of the secret parameter in the *new\_working\_state*.
2. *reseed\_counter*: The re-initialized block counter for reseeding.

#### Dual\_EC\_DRBG Reseed Process

Comment: **pad8** returns a copy of  $s$  padded on the right with binary 0's, if necessary, to a multiple of 8.

1.  $seed\_material = \text{pad8}(s) \parallel \text{entropy\_input} \parallel \text{additional\_input\_string}.$
2.  $s = \text{Hash\_df}(seed\_material, seedlen).$
3.  $reseed\_counter = 0.$
4. Return  $s$  and  $reseed\_counter$  for the *new\_working\_state*.

#### 10.4.2.2.4 Generating Pseudorandom Bits Using Dual\_EC\_DRBG

Notes for the generate function specified in Section 9.4:

The generation of pseudorandom bits using a **Dual\_EC\_DRBG** instantiation requires a call to the generate function specified in Section 9.4. Process step 8 of that function calls the

generate algorithm specified in this section. The values for *max\_number\_of\_bits\_per\_request* and *max\_outlen* are provided in Table 4 of Section 10.4.2.1. *outlen* is the number of pseudorandom bits taken from each *x*-coordinate as the **Dual\_EC\_DRBG** steps. For performance reasons, the value of *outlen* should be set to the maximum value as provided in Table 5. However, an implementation may set *outlen* to any multiple of 8 bits less than or equal to *max\_outlen*. The bits that become the **Dual\_EC\_DRBG** output are always the rightmost bits, i.e., the least significant bits of the *x*-coordinates. Annex C contains additional information regarding the statistical and distributional implications related to the truncation of the *x*-coordinates.

The generate algorithm:

Let **Hash\_df** be the hash derivation function specified in Section 10.5.2 using an appropriate hash function from Table 4 in Section 10.4.2.1. The value of *reseed\_interval* is also provided in Table 4.

The following are used by the generate algorithm:

- a. **pad8** (*bitstring*) returns a copy of the *bitstring* padded on the right with binary 0's, if necessary, to a multiple of 8.
- b. **Truncate** (*bitstring*, *in\_len*, *out\_len*) inputs a *bitstring* of *in\_len* bits, returning a string consisting of the leftmost *out\_len* bits of *bitstring*. If *in\_len* < *out\_len*, the *bitstring* is padded on the right with (*out\_len* - *in\_len*) zeroes, and the result is returned.
- c.  $x(A)$  is the *x*-coordinate of the point *A* on the curve, given in affine coordinates. An implementation may choose to represent points internally using other coordinate systems; for instance, when efficiency is a primary concern. In this case, a point **shall** be translated back to affine coordinates before  $x()$  is applied.
- d.  $\phi(x)$  maps field elements to non-negative integers, taking the bit vector representation of a field element and interpreting it as the binary expansion of an integer.

The precise definition of  $\phi(x)$  used in steps 6 and 7 of the generate process below depends on the field representation of the curve points. In keeping with the convention of FIPS 186-2, the following elements will be associated with each other (note that, in this case, *m* denotes the size of the base field):

*B*:  $c_{m-1} \parallel c_{m-2} \parallel \dots \parallel c_1 \parallel c_0$ , a bitstring, with  $c_{m-1}$  being leftmost.

*Z*:  $c_{m-1}2^{m-1} + \dots + c_22^2 + c_12^1 + c_0 \in \mathbb{Z}$ ;

*Fa*:  $c_{m-1}2^{m-1} + \dots + c_22^2 + c_12^1 + c_0 \bmod p \in F_p$ ;

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

- e. \* is the symbol representing scalar multiplication of a point on the curve.

The following process or its equivalent **shall** be used to generate pseudorandom bits (see step 8 of the generate process in Section 9.4):

**Dual\_EC\_DRBG\_Generate\_algorithm** (*working\_state*, *requested\_number\_of\_bits*, *additional\_input*):

1. *working\_state*: The current values for  $s$ , *seedlen*,  $p$ ,  $a$ ,  $b$ ,  $n$ ,  $P$ ,  $Q$ , and *reseed\_counter* (see Section 10.4.2.2.1).
2. *requested\_number\_of\_bits*: The number of pseudorandom bits to be returned to the generate function.
3. *additional\_input*: The additional input string received from the consuming application. If the input of *additional\_input* is not supported by an implementation, then step 2 of the generate process becomes:

$$\text{additional\_input} = 0.$$

Alternatively, generate steps 2 and 9 are omitted, the *additional\_input* term is omitted from step 5, and the "go to step 5" in step 12 is to the step that now sets  $t = s$ .

### Output:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS**, or an indication that a reseed is required before the requested pseudorandom bits can be generated.
2. *returned\_bits*: The pseudorandom bits to be returned to the generate function.
3.  $s$ : The new value for the secret parameter in the new *working\_state*.
4. *reseed\_counter*: The updated block counter for reseeding.

### Dual\_EC\_DRBG Generate Process:

Comment: Check whether a reseed is required.

1. If  $\left( \text{reseed\_counter} + \left\lceil \frac{\text{requested\_number\_of\_bits}}{\text{outlen}} \right\rceil \right) > \text{reseed\_interval}$ , then return an indication that a reseed is required.

Comment: If *additional\_input* is *Null*, set to *seedlen* zeroes; otherwise, **Hash\_df** to *seedlen* bits.

2. If (*additional\_input\_string* = *Null*), then *additional\_input* = 0  
Else *additional\_input* = **Hash\_df**(**pad8**(*additional\_input\_string*), *seedlen*).

Comment: Produce *requested\_no\_of\_bits*, *outlen* bits at a time:

3. *temp* = the *Null* string.
4.  $i = 0$ .
5.  $t = s \oplus \text{additional\_input}$ .

Comment:  $t$  is to be interpreted as a *seedlen*-bit unsigned integer. To be precise,  $t$  should be reduced mod  $n$ ; the operation  $*$  will effect this.

6.  $s = \phi(x(t * P))$ . Comment:  $s$  is a *seedlen*-bit number. Note that the conversion of  $\phi(x)$  is discussed in item d above; this also applies to step 7.
7.  $r = \phi(x(s * Q))$ . Comment:  $r$  is a *seedlen*-bit number.
8.  $temp = temp \parallel (\text{rightmost } outlen \text{ bits of } r)$ .
9.  $additional\_input = 0$  Comment: *seedlen* zeroes; *additional\_input\_string* is added only on the first iteration.
10.  $reseed\_counter = reseed\_counter + 1$ .
11.  $i = i + 1$ .
12. If  $(len(temp) < requested\_number\_of\_bits)$ , then go to step 5.
13.  $returned\_bits = \text{Truncate}(temp, i \times outlen, requested\_number\_of\_bits)$ .
14.  $s = \phi(x(s * P))$ .
15. Return **SUCCESS**,  $returned\_bits$ , and  $s$ , and  $reseed\_counter$  for the *new\_working\_state*.

## 10.5 Auxilliary Functions

### 10.5.1 Discussion

Derivation functions are internal functions that are used during DRBG instantiation and reseeding to either derive internal state values or to distribute entropy throughout a bitstring. Two methods are provided. One method is based on hash functions (see Section 10.5.2), and the other method is based on block cipher algorithms (see 10.5.3). The block cipher derivation function uses a **Block\_Cipher\_Hash** function that is specified in Section 10.5.4.

The presence of these derivation functions in this Standard does not implicitly approve these functions for any other application.

### 10.5.2 Derivation Function Using a Hash Function (Hash\_df)

This derivation function is used by the **Dual\_EC\_DRBG** specified Section 10.4.2. 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 mechanism, and let *outlen* be its output length.

The following or an equivalent process **shall** be used to derive the requested number of bits.

**Hash\_df** (*input\_string*, *no\_of\_bits\_to\_return*):

1. *input\_string*: The string to be hashed.
2. *no\_of\_bits\_to\_return*: The number of bits to be returned by **Hash\_df**. The maximum length (*max\_number\_of\_bits*) is implementation dependent, but **shall** be less than or equal to  $(255 \times outlen)$ . *no\_of\_bits\_to\_return* is represented as a 32-bit integer.

**Output:**

1. *status*: The status returned from **Hash\_df**. The status will indicate **SUCCESS** or **ERROR\_FLAG**.
2. *requested\_bits* : The result of performing the **Hash\_df**.

**Hash\_df Process:**

1. *temp* = the Null string.
2.  $len = \left\lceil \frac{no\_of\_bits\_to\_return}{outlen} \right\rceil$ .
3. *counter* = an 8-bit binary value representing the integer "1".
4. For *i* = 1 to *len* do

Comment : In step 5.1, *no\_of\_bits\_to\_return* is used as a 32-bit string.

- 4.1 *temp* = *temp* || **Hash** (*counter* || *no\_of\_bits\_to\_return* || *input\_string*).
- 4.2 *counter* = *counter* + 1.
5. *requested\_bits* = Leftmost (*no\_of\_bits\_to\_return*) of *temp*.
6. Return **SUCCESS** and *requested\_bits*.

**10.5.3 Derivation Function Using a Block Cipher Algorithm (Block\_Cipher\_df)**

This derivation function is used by the **CTR\_DRBG** that is specified in Section 10.3.2. Let **BCC** be the function specified in Section 10.5.4. Let *outlen* be its output block length, which is a multiple of 8 bits for the Approved block cipher algorithms, and let *keylen* be the key length.

The following or an equivalent process **shall** be used to derive the requested number of bits.

**Block\_Cipher\_df** (*input\_string*, *no\_of\_bits\_to\_return*) :

1. *input\_string*: The string to be operated on. This string **shall** be a multiple of 8 bits.
2. *no\_of\_bits\_to\_return*: The number of bits to be returned by **Block\_Cipher\_df**. The maximum length (*max\_number\_of\_bits*) is 512 bits for the currently approved block cipher algorithms.

**Output:**

1. *status*: The status returned from **Block\_Cipher\_df**. The status will indicate **SUCCESS** or **ERROR\_FLAG**.
2. *requested\_bits*: The result of performing the **Block\_Cipher\_df**.

**Block\_Cipher\_df Process:**

1. If (*number\_of\_bits\_to\_return* > *max\_number\_of\_bits*), then return an **ERROR\_FLAG**.



2.  $L = \text{len}(\text{input\_string})/8$ .  
Comment:  $L$  is the bitstring representation of the integer resulting from  $\text{len}(\text{input\_string})/8$ .  $L$  **shall** be represented as a 32-bit integer.
3.  $N = \text{number\_of\_bits\_to\_return}/8$ .  
Comment:  $N$  is the bitstring representation of the integer resulting from  $\text{number\_of\_bits\_to\_return}/8$ .  $N$  **shall** be represented as a 32-bit integer.  
Comment: Prepend the string length and the requested length of the output to the *input\_string*.
4.  $S = L \parallel N \parallel \text{input\_string} \parallel 0x80$ .  
Comment: Pad  $S$  with zeros, if necessary.
5. While  $(\text{len}(S) \bmod \text{outlen}) \neq 0$ ,  $S = S \parallel 0x00$ .  
Comment: Compute the starting value.
6.  $\text{temp}$  = the Null string.
7.  $i = 0$ .  
Comment:  $i$  **shall** be represented as a 32-bit integer, i.e.,  $\text{len}(i) = 32$ .
8.  $K$  = Leftmost *keylen* bits of  $0x00010203...1D1E1F$ .
9. While  $\text{len}(\text{temp}) < \text{keylen} + \text{outlen}$ , do
  - 9.1  $IV = i \parallel 0^{\text{outlen} - \text{len}(i)}$ .  
Comment: The 32-bit integer representation of  $i$  is padded with zeros to *outlen* bits.
  - 9.2  $\text{temp} = \text{temp} \parallel \text{BCC}(K, (IV \parallel S))$ .
  - 9.3  $i = i + 1$ .  
Comment: Compute the requested number of bits.
10.  $K$  = Leftmost *keylen* bits of  $\text{temp}$ .
11.  $X$  = Next *outlen* bits of  $\text{temp}$ .
12.  $\text{temp}$  = the Null string.
13. While  $\text{len}(\text{temp}) < \text{number\_of\_bits\_to\_return}$ , do
  - 13.1  $X = \text{Block\_Encrypt}(K, X)$ .
  - 13.2  $\text{temp} = \text{temp} \parallel X$ .
14.  $\text{requested\_bits}$  = Leftmost *number\_of\_bits\_to\_return* of  $\text{temp}$ .
15. Return **SUCCESS** and *requested\_bits*.

#### 10.5.4 BCC Function

**Block\_Encrypt** is used for convenience in the specification of the **BCC** function. This function is not specifically defined in this Standard, but has the following meaning:

**Block\_Encrypt:** A basic encryption operation that uses the selected block cipher algorithm. The function call is:

$$\text{output\_block} = \text{Block\_Encrypt}(\text{Key}, \text{input\_block})$$

For TDEA, the basic encryption operation is called the forward cipher operation (see ANS X9.52); for AES, the basic encryption operation is called the cipher operation (see ASC X9 Registry 00002). The basic encryption operation is equivalent to an encryption operation on a single block of data using the ECB mode.

For the **BCC** function, 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.

**BCC (Key, data) :**

1. *Key*: The key to be used for the block cipher operation.
2. *data*: The data to be operated upon. Note that the length of *data* must be a multiple of *outlen*. This is guaranteed by **Block\_Cipher\_df** process steps 4 and 8.1 in Section 10.5.3.

**Output:**

1. *output\_block*: The result to be returned from the **Block\_Cipher\_Hash** operation.

**BCC Process:**

1.  $\text{chaining\_value} = 0^{\text{outlen}}$ . Comment: Set the first chaining value to *outlen* zeros.
2.  $n = \text{len}(\text{data})/\text{outlen}$ .
3. Starting with the leftmost bits of data, split the *data* into *n* blocks of *outlen* bits, each forming *block*<sub>1</sub> to *block*<sub>*n*</sub>.
4. For *i* = 1 to *n* do
  - 4.1  $\text{input\_block} = \text{chaining\_value} \oplus \text{block}_i$ .
  - 4.2  $\text{chaining\_value} = \text{Block\_Encrypt}(\text{Key}, \text{input\_block})$ .
5. *output\_block* = chaining\_value.
6. Return *output\_block*.

## 11 Assurance

### 11.1 Overview

A user of a DRBG for cryptographic purposes requires assurance that the generator actually produces random and unpredictable bits. The user needs assurance that the design of the generator, its implementation and its use to support cryptographic services are adequate to protect the user's information. In addition, the user requires assurance that the generator continues to operate correctly. The assurance strategy for the DRBG mechanisms in this Standard is depicted in Figure 13.

The design of each DRBG mechanism in this Standard has received an evaluation of its security properties prior to its selection for inclusion in this Standard.

The accuracy of an implementation of a DRBG process **may** be asserted by an implementer. However, this Standard requires that an implementation **shall** be designed to allow validation testing, including documenting design assertions about how the DRBG mechanism operates (see Section 11.2). This **shall** include mechanisms for testing all detectable error conditions.

An implementation **should** be validated for conformance to this Standard (see Section 11.3). The consuming application or cryptographic service that uses a DRBG mechanism **should** also be validated and periodically tested for continued correct operation. However, this level of testing is outside the scope of this Standard. Such validations provide a higher level of assurance that the DRBG mechanism is correctly implemented. Validation testing for DRBG mechanisms consists of testing whether or not the DRBG mechanism produces the expected result, given a specific set of input parameters (e.g., entropy input). Implementations used directly by consuming applications **should** also be validated against conformance to FIPS 140-2.

Health tests on the DRBG mechanism **shall** be implemented within a DRBG mechanism boundary or sub-boundary in order to determine that the process continues to operate as designed and implemented. See Section 11.4 for further information.

Note that any entropy input used for testing (either for validation testing or health testing) may be publicly known. Therefore, entropy input used for testing **shall not** knowingly be used for normal operational use.

### 11.2 Minimal Documentation Requirements

A set of documentation **shall** be developed that will provide assurance to users and (optionally) validators that the DRBG mechanisms in this Standard have been implemented properly. Much of this documentation may be placed in a user's manual. This documentation **shall** consist of the following as a minimum:

- Document the method for obtaining entropy input.

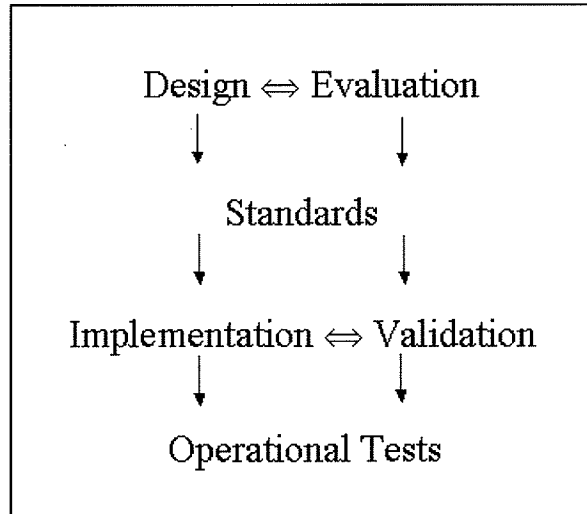


Figure 13: DRBG Assurance Strategy

- Document how the implementation has been designed to permit implementation validation and health testing.
- Document the type of DRBG mechanism (e.g., HMAC\_DRBG, Dual\_EC\_DRBG), and the cryptographic primitives used (e.g., SHA-256).
- Document the security strengths supported by the implementation.
- Document features supported by the implementation (e.g., prediction resistance, the available elliptic curves, etc.).
- If DRBG mechanism functions are distributed, specify the mechanisms that are used to protect the confidentiality and integrity of the internal state or parts of the internal state that are transferred between the distributed DRBG mechanism sub-boundaries.
- In the case of the **CTR\_DRBG**, indicate whether a derivation function is provided. If a derivation function is not used, document that the implementation can only be used if full entropy input is available:
- Document any support functions other than health testing.
- Document the periodic intervals at which health testing is performed for the generate function and provide a justification for the selected intervals (see Section 11.4.4).
- Document how the integrity of the health tests will be determined subsequent to implementation validation.

### **11.3 Implementation Validation Testing**

A DRBG mechanism **shall** be tested for conformance to this Standard. Regardless of whether or not validation testing is obtained by an implementer, a DRBG mechanism **shall** be designed to be tested to ensure that the product is correctly implemented; this will allow validation testing to be obtained by a consumer, if desired. A testing interface **shall** be available for this purpose in order to allow the insertion of input and the extraction of output for testing.

Implementations to be validated **shall** include the following:

- Documentation specified in Section 11.2.
- Any documentation or results required for validation testing.

### **11.4 Health Testing**

#### **11.4.1 Overview**

A DRBG implementation **shall** perform self-tests to obtain assurance that the DRBG continues to operate as designed and implemented (health testing). The testing function(s) within a DRBG mechanism boundary (or sub-boundary) **shall** test each DRBG mechanism function within that boundary (or sub-boundary), with the possible exception of the test function itself. Note that testing may require the creation and use of an instantiation for testing purposes only. A DRBG implementation may optionally perform other self-tests for DRBG functionality in addition to the tests specified in this Standard.

All data output from the DRBG mechanism boundary (or sub-boundary) **shall** be inhibited while these tests are performed. The results from known-answer-tests (see Section 11.4.2) **shall not** be output as random bits during normal operation.

#### 11.4.2 Known-Answer Testing

Known-answer testing **shall** be conducted as specified below. A known-answer test involves operating the DRBG mechanism with 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 mechanism **shall** enter an error state and output an error indicator (see Section 11.4.7).

Generalized known-answer testing is specified in Sections 11.4.3 to 11.4.6. Testing **shall** be performed on all implemented DRBG mechanism functions, with the possible exception of the test function itself. Documentation **shall** be provided that addresses the continued integrity of the health tests (see Section 11.2).

#### 11.4.3 Testing the Instantiate Function

Known-answer tests on the instantiate function **shall** be performed prior to creating each operational instantiation. However, if several instantiations are performed in quick succession using the same *security\_strength* and *prediction\_resistance\_flag* parameters, then the testing may be reduced to testing only prior to creating the first instantiation using that parameter set until such time as the succession of instantiations is completed. Thereafter, other instantiations **shall** be tested as specified above.

The *security\_strength* and *prediction\_resistance\_flag* to be used in the operational invocation **shall** be used during the test. Representative fixed values and lengths of the *entropy\_input*, *nonce* and *personalization\_string* (if supported) **shall** be used; the value of the *entropy\_input* used during testing **shall not** be intentionally reused during normal operations (either by the instantiate or the reseed functions). Error handling **shall** also be tested, including whether or not the instantiate function handles an error from the source of entropy input correctly.

If the values used during the test produce the expected results, and errors are handled correctly, then the instantiate function may be used to instantiate using the tested values of *security\_strength* and *prediction\_resistance\_flag*.

An implementation **should** provide a capability to test the instantiate function on demand.

#### 11.4.4 Testing the Generate Function

Known-answer tests **shall** be performed on the generate function before the first use of the function in an implementation (i.e., the first use ever) and at reasonable intervals defined by the implementer. The implementer **shall** document the intervals and provide a justification for the selected intervals.

The known-answer tests **shall** be performed for each implemented *security\_strength*. Representative fixed values and lengths for the *requested\_number\_of\_bits* and *additional\_input* (if supported) and the working state of the internal state value (see Sections 8.2.3 and 10) **shall** be used. If prediction resistance is supported, then each combination of the *security\_strength*, *prediction\_resistance\_request* and *prediction\_resistance\_flag* **shall** be tested. The error handling for each input parameter **shall** also be tested, and testing **shall** include setting the *reseed\_counter* to meet or exceed the *reseed\_interval* in order to check that the implementation is reseeded or that the DRBG is "shut down", as appropriate.

If the values used during the test produce the expected results, and errors are handled correctly, then the generate function may be used during normal operations.

Bits generated during health testing **shall not** be output as pseudorandom bits.

An implementation **should** provide a capability to test the generate function on demand.

#### **11.4.5 Testing the Reseed Function**

A known-answer test of the reseed function **shall** use the *security\_strength* in the internal state of the instantiation to be reseeded. Representative values of the *entropy\_input* and *additional\_input* (if supported) and the working state of the internal state value shall be used (see Sections 8.2.3 and 10). Error handling shall also be tested, including an error in obtaining the *entropy\_input* (e.g., the *entropy\_input* source is broken).

If the values used during the test produce the expected results, and errors are handled correctly, then the reseed function may be used to reseed the instantiation.

Self-test **shall** be performed as follows:

1. When prediction resistance is supported in an implementation, the reseed function **shall** be tested whenever the generate function is tested (see above).
2. When prediction resistance is not supported in an implementation, the reseed function **shall** be tested whenever the reseed function is invoked and before the reseed is performed on the operational instantiation.

An implementation **should** provide a capability to test the reseed function on demand.

#### **11.4.6 Testing the Uninstantiate Function**

The uninstantiate function **shall** be tested whenever other functions are tested. Testing **shall** attempt to demonstrate that error handling is performed correctly, and the internal state has been erased.

#### **11.4.7 Error Handling**

##### **11.4.7.1 General Discussion**

The expected errors are indicated for each DRBG mechanism function (see Sections 9.2 - 9.5) and for the derivation functions in Section 10.5. The error handling routines **should** indicate the type of error.

##### **11.4.7.2 Errors Encountered During Normal Operation**

Many errors during normal operation may be caused by a consuming application's improper DRBG request; these errors are indicated by **ERROR\_FLAG** in the pseudocode. In these cases, the consuming application user is responsible for correcting the request within the limits of the user's organizational security policy. For example, if a failure indicating an invalid requested security strength is returned, a security strength higher than the DRBG or the DRBG instantiation can support has been requested. The user may reduce the requested security strength if the organization's security policy allows the information to be protected using a lower security strength, or the user **shall** use an appropriately instantiated DRBG.

Catastrophic errors (i.e., those errors indicated by the **CATASTROPHIC\_ERROR\_FLAG** in the pseudocode) detected during normal operation **shall** be treated in the same manner as an error detected during health testing (see Section 11.4.7.3).

#### **11.4.7.3 Errors Encountered During Health Testing**

Errors detected during health testing **shall** be perceived as catastrophic DRBG failures.

When a DRBG fails a health test or a catastrophic error is detected during normal operation, the DRBG **shall** enter an error state and output an error indicator. The DRBG **shall not** perform any DRBG operations while in the error state, and pseudorandom bits **shall not** be output when an error state exists. When in an error state, user intervention (e.g., power cycling of the DRBG) **shall** be required to exit the error state, and the DRBG **shall** be re-instantiated before the DRBG can be used to produce pseudorandom bits. Examples of such behavior include:

- A test deliberately inserts an error, and the error is not detected, or
- A different result is returned from the instantiate, reseed, generate or uninstantiate function than was expected.

## Annex A: (Normative) Application-Specific Constants

### A.1 Constants for the Dual\_EC\_DRBG

The **Dual\_EC\_DRBG** requires the specifications of an elliptic curve and two points on the elliptic curve. One of the following curves and with associated points **shall** be used in applications requiring certification under ASC X9 Registry 00001. More details about these curves may be found in FIPS PUB 186-3, the Digital Signature Standard [1].

#### A.1.1 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 for consistency with FIPS 186-3 but is referred to as  $n$  in the description of the **Dual\_EC\_DRBG**.

$a$  - (-3) in the above equation.

$b$  - coefficient above.

The  $x$  and  $y$  coordinates of the base point, ie generator  $G$ , are the same as for the point  $P$ .

##### A.1.1.1 Curve P-256

$p$  = 11579208921035624876269744694940757353008614\  
3415290314195533631308867097853951

$r$  = 11579208921035624876269744694940757352999695\  
5224135760342422259061068512044369

$b$  = 5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f6 3bce3c3e  
27d2604b

$P_x$  = 6b17d1f2 e12c4247 f8bce6e5 63a440f2 77037d81 2deb33a0  
f4a13945 d898c296

$P_y$  = 4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16 2bce3357 6b315ece  
cbb64068 37bf51f5



## Draft ANS X9.82, Part 3 - November 2006

$Qx$  = c97445f4 5cdef9f0 d3e05e1e 585fc297 235b82b5 be8ff3ef  
ca67c598 52018192

$Qy$  = b28ef557 ba31dfcb dd21ac46 e2a91e3c 304f44cb 87058ada  
2cb81515 1e610046

### A.1.1.2 Curve P-384

$p$  = 39402006196394479212279040100143613805079739\  
27046544666794829340424572177149687032904726\  
6088258938001861606973112319

$r$  = 39402006196394479212279040100143613805079739\  
27046544666794690527962765939911326356939895\  
6308152294913554433653942643

$b$  = 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 0a60blce 1d7e819d 7a431d7c 90ea0e5f

$Qx$  = 8e722de3 125bddb0 5580164b fe20b8b4 32216a62 926c5750  
2ceede31 c47816ed d1e89769 124179d0 b6951064 28815065

$Qy$  = 023b1660 dd701d08 39fd45ee c36f9ee7 b32e13b3 15dc0261  
0aalb636 e346df67 1f790f84 c5e09b05 674dbb7e 45c803dd

### A.1.1.3 Curve P-521

$p$  = 68647976601306097149819007990813932172694353\  
00143305409394463459185543183397656052122559\  
64066145455497729631139148085803712198799971\  
6643812574028291115057151

$r$  = 68647976601306097149819007990813932172694353\  
00143305409394463459185543183397655394245057\  
74633321719753296399637136332111386476861244\  
0380340372808892707005449

$b =$  051953eb 9618e1c9 a1f929a2 1a0b6854 0eea2da7 25b99b31 5f3b8b48  
9918ef10 9e156193 951ec7e9 37b1652c 0bd3bb1b f073573d f883d2c3  
4f1ef451 fd46b503 f00

$P_x =$  c6858e06 b70404e9 cd9e3ecb 662395b4 429c6481 39053fb5  
21f828af 606b4d3d baa14b5e 77efe759 28fel1dc1 27a2ffa8  
de3348b3 c1856a42 9bf97e7e 31c2e5bd 66

$P_y =$  11839296 a789a3bc 0045c8a5 fb42c7d1 bd998f54 449579b4  
46817afb d17273e6 62c97ee7 2995ef42 640c550b 9013fad0  
761353c7 086a272c 24088be9 4769fd16 650

$Q_x =$  1b9fa3e5 18d683c6 b6576369 4ac8efba ec6fab44 f2276171  
a4272650 7dd08add 4c3b3f4c 1ebc5b12 22ddba07 7f722943  
b24c3edf a0f85fe2 4d0c8c01 591f0be6 f63

$Q_y =$  1f3bdba5 85295d9a 1110d1df 1f9430ef 8442c501 8976ff34  
37ef91b8 1dc0b813 2c8d5c39 c32d0e00 4a3092b7 d327c0e7  
a4d26d2c 7b69b58f 90666529 11e45777 9de

## A.2 Using Alternative Points in the Dual\_EC\_DRBG()

The security of **Dual\_EC\_DRBG** requires that the points  $P$  and  $Q$  be properly generated. To avoid using potentially weak points, the points specified in Annex A.1 **should** be used. However, an implementation may use different pairs of points provided that they are verifiably random, as evidenced by the use of the procedure specified in Annex A.2.1 below, and the self-test procedure in Annex A.2.2. An implementation that uses alternative points generated by this Approved method **shall** have them “hard-wired” into its source code, or hardware, as appropriate, and loaded into the *working\_state* at instantiation. To conform to this Standard, alternatively generated points **shall** use the procedure given in Annex A.2.1, and verify their generation using Annex A.2.2.

### A.2.1 Generating Alternative $P, Q$

The curve **shall** be one of the curves that is specified in Annex A.1 of this Standard, and **shall** be appropriate for the desired *security\_strength*, as specified in Table 4, Section 10.4.2.1.

The points  $P$  and  $Q$  **shall** be valid base points for the selected elliptic curve that are generated to be verifiably random using the procedure specified in ANS X9.62. The following input is required for each point:

An elliptic curve  $E = (F_p, a, b)$ , cofactor  $h$ , prime  $n$ , a bit string *domain\_parameter\_seed*<sup>3</sup>, and hash function **Hash**(). The curve parameters are given in Annex A of this Standard. The *domain\_parameter\_seed* **shall** be different for each point, and the minimum length  $m$  of each *domain\_parameter\_seed* **shall** conform to Section 10.4.1, Table 4, under “Seed length”. The

---

<sup>3</sup> Called a *SEED* in ANS X9.62.

bit length of the *domain\_parameter\_seed* may be larger than  $m$ . The hash function **shall** be SHA-512 in all cases.

If the output from the ANS X9.62 generation procedure is "failure", a different *domain\_parameter\_seed* **shall** be used for the point being generated.

Otherwise, the output from the generate procedure in ANS 9.62 **shall** be used.

### **A.2.2 Additional Self-testing Required for Alternative $P, Q$**

To insure that the points  $P$  and  $Q$  have been generated appropriately, additional self-test procedures **shall** be performed whenever the instantiate function is invoked. Section 11.4.2 specifies that known-answer tests on the instantiate function be performed prior to creating an operational instantiation. As part of these tests, an implementation of the generation procedure specified in ANS X9.62 **shall** be called for each point (i.e.,  $P$  and  $Q$ ) with the appropriate *domain\_parameter\_seed* value that was used to generate that point. The point returned **shall** be compared with the corresponding stored value of the point. If the generated value does not match the stored value, the implementation **shall** halt with an error condition.

## ANNEX B : (Normative) Conversion and Auxilliary Routines

### B.1 Bitstring to an Integer

**Bitstring\_to\_integer** ( $b_1, b_2, \dots, b_n$ ):

1.  $b_1, b_2, \dots, b_n$  The bitstring to be converted.

**Output:**

1.  $x$  The requested integer representation of the bitstring.

**Process:**

1. Let  $(b_1, b_2, \dots, b_n)$  be the bits of  $b$  from leftmost to rightmost.
2. 
$$x = \sum_{i=1}^n 2^{(n-i)} b_i .$$
3. Return  $x$ .

In this Standard, the binary length of an integer  $x$  is defined as the smallest integer  $n$  satisfying  $x < 2^n$ .

### B.2 Integer to a Bitstring

**Integer\_to\_bitstring** ( $x$ ):

1.  $x$  The non-negative integer to be converted.

**Output:**

1.  $b_1, b_2, \dots, b_n$  The bitstring representation of the integer  $x$ .

**Process:**

1. Let  $(b_1, b_2, \dots, b_n)$  represent the bitstring, where  $b_1 = 0$  or  $1$ , and  $b_1$  is the most significant bit, while  $b_n$  is the least significant bit.
2. For any integer  $n$  that satisfies  $x < 2^n$ , the bits  $b_i$  **shall** satisfy:

$$x = \sum_{i=1}^n 2^{(n-i)} b_i .$$

3. Return  $b_1, b_2, \dots, b_n$ .

In this Standard, the binary length of the integer  $x$  is defined as the smallest integer  $n$  that satisfies  $x < 2^n$ .

### B.3 Integer to a Byte String

**Integer\_to\_byte\_string** ( $x$ ):

1. A non-negative integer  $x$ , and the intended length  $n$  of the byte string satisfying

$$2^{8n} > x.$$

**Output:**

1. A byte string  $O$  of length  $n$  bytes.

**Process:**

1. Let  $O_1, O_2, \dots, O_n$  be the bytes of  $O$  from leftmost to rightmost.
2. The bytes of  $O$  **shall** satisfy:

$$x = \sum 2^{8(n-i)} O_i$$

for  $i = 1$  to  $n$ .

3. Return  $O$ .

#### B.4 Byte String to an Integer

**Byte\_string\_to\_integer ( $O$ ):**

1. A byte string  $O$  of length  $n$  bytes.

**Output:**

1. A non-negative integer  $x$ .

**Process:**

1. Let  $O_1, O_2, \dots, O_n$  be the bytes of  $O$  from leftmost to rightmost.
2.  $x$  is defined as follows:

$$x = \sum 2^{8(n-i)} O_i$$

for  $i = 1$  to  $n$ .

3. Return  $x$ .

## Annex C: (Informative) Security Considerations

### C.1 Extracting Bits in the Dual\_EC\_DRBG (...)

#### C.1.1 Potential Bias Due to Modular Arithmetic for Curves Over $F_p$

Given an integer  $x$  in the range 0 to  $2^N-1$ , where  $N$  is any positive integer, the  $r^{th}$  bit of  $x$  depends solely upon whether  $\left\lfloor \frac{x}{2^r} \right\rfloor$  is odd or even. Exactly  $\frac{1}{2}$  of the integers in this range have the property that their  $r^{th}$  bit is 0. But if  $x$  is restricted to  $F_p$ , i.e., to the range 0 to  $p-1$ , this statement is no longer true.

By excluding the  $k = 2^N - p$  values  $p, p+1, \dots, 2^N-1$  from the set of all integers in  $Z_N$ , the ratio of ones to zeroes in the  $r^{th}$  bit is altered from  $2^{N-1} / 2^{N-1}$  to a value that can be no smaller than  $(2^{N-1} - k) / 2^{N-1}$ . For all the primes  $p$  used in this Standard,  $k/2^{N-1}$  is smaller than  $2^{-31}$ . Thus, the ratio of ones and zeroes in any bit is within at least  $2^{-31}$  of 1.0.

To detect this small difference from random, a sample of at least  $2^{64}$  outputs is required before the observed distribution of 1's and 0's is more than one standard deviation away from flat random. This effect is dominated by the bias addressed below in Annex C.1.2.

#### C.1.2 Adjusting for the Missing Bit(s) of Entropy in the $x$ Coordinates.

In a truly random sequence, it should not be possible to predict any bits from previously observed bits. With the **Dual\_EC\_DRBG**, the full output block of bits produced by the algorithm is "missing" some entropy. Fortunately, by discarding some of the bits, those bits remaining can be made to have nearly "full strength", in the sense that the entropy that they are missing is negligibly small.

To illustrate what can happen, suppose that the curve with P-256 is selected, and that all 256 bits produced were output by the generator, i.e. that *outlen* = 256 also. Suppose also that 255 of these bits are published, and the 256<sup>th</sup> bit is kept "secret". About  $\frac{1}{2}$  the time, the unpublished bit could easily be determined from the other 255 bits. Similarly, if 254 of the bits are published, about  $\frac{1}{4}$  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  $2^m$  bitstrings of length  $m$  occur in the list of all  $x$  coordinates of curve points.

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  $2 * p^{1/2}$  of  $p$ ). For the curves in Annex A.1, 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  $f = 1$  is the curve's cofactor. For all the curves in this Standard, the cofactor  $f = 1$ .

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

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

For each  $0 \leq j \leq 2^d$ , the term in braces represents the approximate number of bitstrings  $b$  of length  $(m-d)$  such that there are exactly  $j$  points whose  $x$ -coordinates have their  $(m-d)$  least significant bits equal to  $b$ ;  $z = (2f-1)/2f$  is the probability that any particular string occurs in an  $x$  coordinate;  $p_j = (j*2f)/2^m$  is the probability that a member of the  $j^{\text{th}}$  category occurs. Note that the  $j=0$  category contributes nothing to the entropy (randomness).

The values of  $E$  for  $d$  up to 16 are:

$\log_2(f)$ :	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

The analysis above uses Shannon entropy. As discussed elsewhere in this Standard, min-entropy is a more appropriate measure of randomness than Shannon entropy, at least for the purposes of security. If the analysis above is repeated for min-entropy, then one finds that about 1 bit of min-entropy is missing for most values of  $d < m/2$ . The main reason for this is that the case of  $j = 2^d$  is expected to occur, provided that  $d < m/2$ . Therefore the maximum probability for a particular bit string of length  $m-d$  is  $p_{2^d} = 2^{d+1-m}$ , which gives a min-entropy of  $m-d-1$ . An adversary attempting to guess the value of the bit string of length  $m-d$ , would choose a string such that  $j = 2^d$ . On the other hand, generally speaking, the security strength associated with an  $m$ -bit elliptic curve is only  $m$  bits, which implies that only  $m/2$  bits of min-entropy are required. Therefore, the loss of a single bit of min-entropy may be deemed acceptable here because the min-entropy would still be well over what is needed.

Observations:

- a) The table starts where it should, at 1 missing bit;
- b) The missing entropy rapidly decreases;
- c) For the curves in this Standard,  $d=13$  leaves 1 bit of information in every 10,000  $(m-13)$ -bit outputs (i.e., one bit of entropy is missing in a collection of 10,000 outputs).

Based on these calculations, for the mod  $p$  curves, it is recommended that an implementation **shall** remove at least the **leftmost** (most significant) 13 bits of every  $m$ -bit output.

For ease of implementation, the value of  $d$  **should** be adjusted upward, if necessary, until the number of bits remaining,  $m-d = \text{outlen}$ , is a multiple of 8. By this rule, the recommended number of bits discarded from each  $x$ -coordinate will be either 16 or 17. As noted in Section 10.4.2.2.4, an implementation may decide to truncate additional bits from each  $x$ -coordinate, provided that the number retained is a multiple of 8.

Because only half of all values in  $[0, 1, \dots, p-1]$  are valid  $x$ -coordinates on an elliptic curve defined over  $\mathbb{F}_p$ , it is clear that full  $x$ -coordinates **should not** be used as pseudorandom bits. The solution to this problem is to truncate these  $x$ -coordinates by removing the high order 16 or 17 bits. The entropy loss associated with such truncation amounts has been demonstrated to be minimal (see the above chart).

When 16 high-order bits of a random  $x$ -coordinate have been removed, an adversary that is given the remaining bits has a probability of about  $\frac{1}{2} + 1/641$  of guessing correctly whether the bits given are truly random or derived from a random  $x$ -coordinate. Shannon entropy of 239.999989 out of a maximum of 240 does not guarantee indistinguishability. One 90,000<sup>th</sup> of a bit missing does not translate into a 1/90,000 advantage for a distinguishing adversary, but rather a 1/640 advantage. When it is crucial that an adversary cannot distinguish the DRBG output from a random output, such as when it is used as a stream cipher, then more bits should be truncated from the  $x$ -coordinate, accordingly.

One might wonder if it would be desirable to truncate more than this amount. The obvious drawback to such an approach is that increasing the truncation amount hinders the performance. However, there is an additional reason that argues against increasing the truncation. Consider the case where the low  $s$  bits of each  $x$ -coordinate are kept. Given some subinterval  $I$  of length  $2^s$  contained in  $[0, p)$ , and letting  $N(I)$  denote the number of  $x$ -coordinates in  $I$ , recent results on the distribution of  $x$ -coordinates in  $[0, p)$  provide the following bound:

$$\left| \frac{N(I)}{(p/2)} - \frac{2^s}{p} \right| < \frac{k * \log^2 p}{\sqrt{p}},$$

where  $k$  is some constant derived from the asymptotic estimates given in [2]. For the case of P-521, this is roughly equivalent to:

$$|N(I) - 2^{(s-1)}| < k * 2^{277},$$

where the constant  $k$  is independent of the value of  $s$ . For  $s < 2^{277}$ , this inequality is weak and provides very little support for the notion that these truncated  $x$ -coordinates are uniformly distributed. On the other hand, the larger the value of  $s$ , the sharper this inequality becomes, providing stronger evidence that the associated truncated  $x$ -coordinates are uniformly distributed. Therefore, by keeping truncation to an acceptable minimum, the performance is increased, and certain guarantees can be made about the uniform distribution of the resulting truncated quantities. Further discussion of the uniformity of the truncated  $x$ -coordinates is found in [3], where the form of the prime defining the field is also taken into account.





## ANNEX D: (Informative) DRBG Mechanism Selection

### D.1 Choosing a DRBG Algorithm

Almost no application or system designer starts with the primary purpose of generating good random bits. Instead, the designer typically starts with a goal that he wishes to accomplish, then decides on some cryptographic mechanisms, such as digital signatures or block ciphers that can help achieve that goal. Typically, as the requirements of those cryptographic mechanisms are better understood, he learns random bits will need to be generated, and that this must be done with great care so that the cryptographic mechanisms will be weakened. At this point, there are three things that may guide the designer's choice of a DRBG mechanism:

- a. He may already have decided to include a set of cryptographic primitives as part of his implementation. By choosing a DRBG mechanism based on one of these primitives, he can minimize the cost of adding that DRBG mechanism. In hardware, this translates to lower gate count, less power consumption, and less hardware that must be protected against probing and power analysis. In software, this translates to fewer lines of code to write, test, and validate.

For example, a module that generates RSA signatures has an available hash function, so a hash-based DRBG mechanism is a natural choice.

- b. He may already have decided to trust a block cipher, hash function, keyed hash function, etc., to have certain properties. By choosing a DRBG mechanism based on similar properties, he can minimize the number of algorithms he has to trust.

For example, an AES-based DRBG mechanism might be a good choice when a module provides encryption with AES. Since the security of the module is dependent on the strength of AES, the module's security is not made dependent on any additional cryptographic primitives or assumptions.

- c. Multiple cryptographic primitives may be available within the system or application, but there may be restrictions that need to be addressed (e.g., code size or performance requirements).

For example, a module with support for both hash functions and block ciphers might use the **CTR\_DRBG** if the ability to parallelize the generation of random bits is needed.

The DRBG mechanisms specified in this Standard have different performance characteristics, implementation issues, and security assumptions.

### D.2 HMAC\_DRBG

**HMAC\_DRBG** is built around the use of some approved hash function in the HMAC construction. To generate pseudorandom bits from a secret key (*Key*) and a starting value *V*, the **HMAC\_DRBG** mechanism computes

$$V = \text{HMAC}(\text{Key}, V).$$

At the end of a generation request, the **HMAC\_DRBG** generates a new *Key* and *V*, each requiring one HMAC computation.

**Performance.** **HMAC\_DRBG** produces pseudorandom outputs considerably more slowly than the underlying hash function processes inputs; for SHA-256, a long generate request produces output bits at about 1/4 of the rate that the hash function can process input bits. Each generate request also involves additional overhead equivalent to processing 2048 extra bits with SHA-256. Note, however, that hash functions are typically quite fast; few if any consuming applications are expected to need output bits faster than **HMAC\_DRBG** can provide them.

**Security.** The security of **HMAC\_DRBG** is based on the assumption that an Approved hash function used in the HMAC construction is a pseudorandom function family. Informally, this means that when an attacker doesn't know the key used, HMAC outputs look random, even given knowledge and control over the inputs. In general, even relatively weak hash functions seem to be quite strong when used in the HMAC construction. On the other hand, there is not a reduction proof from the hash function's collision resistance properties to the security of the DRBG; the security of **HMAC\_DRBG** ultimately relies on the pseudorandomness properties of the underlying hash function. Note that the pseudorandomness of HMAC is a widely used assumptions in designs.

**Constraints on Outputs.** As shown in Table 2 of Section 10.2.1, for each hash function, up to  $2^{48}$  generate requests may be made, each of up to  $2^{19}$  bits.

**Resources.** **HMAC\_DRBG** requires access to a dedicated HMAC implementation for optimal performance. However, a general-purpose hash function implementation can always be used to implement HMAC. Any implementation requires the storage space required for the internal state (see Section 10.2.2.1).

**Algorithm Choices.** The choice of hash functions that may be used by **HMAC\_DRBG** is discussed in Section 10.2.1.

### D.3 CTR\_DRBG

**CTR\_DRBG** is based on using an Approved block cipher algorithm in counter mode. At the present time, only three-key TDEA and AES are approved for use in this DRBG mechanism. Pseudorandom outputs are generated by encrypting successive values of a counter; after a generate request, a new key and new starting counter value are generated.

**Performance.** For large Generate requests, **CTR\_DRBG** produces outputs at the same speed as the underlying block cipher algorithm encrypts data. Furthermore, **CTR\_DRBG** is parallelizable. At the end of each Generate request, work equivalent to 2, 3 or 4 encryptions is performed, depending on the choice of underlying block cipher algorithm, to generate new keys and counters for the next Generate request.

**Security.** The security of **CTR\_DRBG** is directly based on the security of the underlying block cipher algorithm, in the sense that, so long as some limits on the total number of outputs are observed, any attack on **CTR\_DRBG** represents an attack on the underlying block cipher algorithm.

**Constraints on Outputs.** As shown in Table 3 of Section 10.3.2.1, for each of the three AES key sizes, up to  $2^{48}$  generate requests may be made, each of up to  $2^{19}$  bits, with a negligible chance of any weakness that does not represent a weakness in AES. However, the smaller block size of TDEA imposes more constraints: each generate request is limited to  $2^{13}$  bits, and at most  $2^{32}$  such requests may be made.

**Resources.** **CTR\_DRBG** may be implemented with or without a derivation function.

When a derivation function is used, **CTR\_DRBG** can process the personalization string and any additional input in the same way as any other DRBG mechanism, but at a cost in performance because of the use of the derivation function (as opposed to not using the derivation function; see below). Such an implementation may be seeded by any Approved source of entropy input that may or may not provide full entropy.

When a derivation function is not used, **CTR\_DRBG** is more efficient when the personalization string and any additional input are provided, but is less flexible because the lengths of the personalization string and additional input cannot exceed *seedlen* bits. Such implementations must be seeded by a source of entropy input that provides full entropy (e.g., an Approved conditioned entropy source or Approved NRBG).

**CTR\_DRBG** requires access to a block cipher algorithm, including the ability to change keys, and the storage space required for the internal state (see Section 10.3.2.2.1).

**Algorithm Choices.** The choice of block cipher algorithms and key sizes that may be used by **CTR\_DRBG** is discussed in Section 10.3.2.1.

#### D.4 DRBGs Based on Hard Problems

The **Dual\_EC\_DRBG** generates pseudorandom outputs by extracting bits from elliptic curve points. The secret, internal state of the DRBG is a value  $s$  that is the  $x$ -coordinate of a point on an elliptic curve. Outputs are produced by first computing  $r$  to be the  $x$ -coordinate of the point  $s*P$  and then extracting low order bits from the  $x$ -coordinate of the elliptic curve point  $r*Q$ .

**Performance.** Due to the elliptic curve arithmetic involved in this DRBG mechanism, this algorithm generates pseudorandom bits more slowly than the other DRBG mechanisms in this Standard. It should be noted, however, that the design of this algorithm allows for certain performance-enhancing possibilities. First, note that the use of fixed base points allows a substantial increase in the performance of this DRBG mechanism via the use of tables. By storing multiples of the points  $P$  and  $Q$ , the elliptic curve multiplication can be accomplished via point additions rather than multiplications, a much less expensive operation. In more constrained environments where table storage is not an option, the use of so-called Montgomery Coordinates of the form  $(X : Z)$  can be used as a method to increase performance, since the  $y$ -coordinates of the computed points are not required. Alternatively, Jacobian or Projective Coordinates of the form  $(X, Y, Z)$  can speed up the elliptic curve operation. These have been shown to be competitive with Montgomery for the NIST curves, and are straightforward to implement.

A given implementation of this DRBG mechanism need not include all three of the curves specified in Annex A.1. Once the designer decides upon the strength required by a given application, he can then choose to implement the single curve that most appropriately meets this requirement. For a common level of optimization expended, the higher strength curves will be slower and tend toward less efficient use of output blocks. To mitigate the latter, the designer should be aware that every distinct request for random bits requires the computational expense of at least two elliptic curve point multiplications.

Applications requiring large blocks of random bits (such as IKE or SSL), can thus be implemented most efficiently by first making a single call to the **Dual\_EC\_DRBG** for all the required bits, and then appropriately partitioning these bits as required by the protocol. For applications that already have hardware or software support for elliptic curve arithmetic, this DRBG mechanism is a natural choice, as it allows the designer to utilize existing capabilities to generate random numbers.

**Security.** The security of **Dual\_EC\_DRBG** is based on the Elliptic Curve Discrete Logarithm Problem that has no known attacks better than the meet-in-the-middle attacks. For an elliptic curve defined over a field of size  $2^m$ , the work factor of these attacks is approximately  $2^{m/2}$ , so that solving this problem is computationally infeasible for the curves in this Standard. The **Dual\_EC\_DRBG** is the only DRBG mechanism in this Standard whose security is related to a hard problem in number theory.

**Constraints on Outputs.** For any one of the three elliptic curves listed in Annex A.1, a particular instance of **Dual\_EC\_DRBG** may generate at most  $2^{32}$  output blocks before reseeding, where the size of the output blocks is discussed in Section 10.4.2.2.4. Since the sequence of output blocks is expected to cycle in approximately  $\sqrt{n}$  bits (where  $n$  is the (prime) order of the particular elliptic curve being used), this is quite a conservative reseed interval for any one of the three possible curves.

**Resources.** Any source of entropy input may be used with **Dual\_EC\_DRBG**, provided that it is capable of generating at least *min\_entropy* bits of entropy in a string of *max\_length* =  $2^{13}$  bits. This DRBG mechanism also requires an appropriate hash function (see Table 4) that is used exclusively for producing an appropriately-sized initial state from the entropy input at instantiation or reseeding. An implementation of this DRBG mechanism must also have enough storage for the internal state (see 10.4.2.2.1). Some optimizations require additional storage for moderate to large tables of pre-computed values.

**Algorithm Choices.** The choice of appropriate elliptic curves and points used by **Dual\_EC\_DRBG** is discussed in Annex A.1.

## D.5 Summary for DRBG Selection

Table D-1 provides a summary of the DRBG mechanisms in this Standard.

**Table 1: DRBG Mechanism Summary**

	Dominating Cost/Block	Constraints (max.)
<b>HMAC_DRBG</b>	4 hash function calls	$2^{48}$ calls of $2^{19}$ bits
<b>CTR_DRBG (TDEA)</b>	1 TDEA encrypt	$2^{32}$ calls of $2^{13}$ bits
<b>CTR_DRBG (AES)</b>	1 AES encrypt	$2^{48}$ calls of $2^{19}$ bits
<b>Dual_EC_DRBG</b>	2 EC points	$2^{32}$ blocks

## ANNEX E: (Informative) Example Pseudocode for Each DRBG Mechanism

### E.1 Preliminaries

The internal states in these examples are considered to be an array of states, identified by *state\_handle*. A particular state is addressed as *internal\_state (state\_handle)*, where the value of *state\_handle* begins at 0 and ends at  $n-1$ , and  $n$  is the number of internal states provided by an implementation. A particular element in the internal state is addressed by *internal\_state (state\_handle).element*. In an empty internal state, all bitstrings are set to *Null*, and all integers are set to 0.

For each example in this annex, arbitrary values have been selected that are consistent with the allowed values for each DRBG mechanism, as specified in the appropriate table in Section 10.

The pseudocode in this annex does not include the necessary conversions (e.g., integer to bitstring) for an implementation. When conversions are required, they must be accomplished as specified in Annex B.

The following routine is defined for these pseudocode examples:

**Find\_state\_space ()**: A function that finds an unused internal state. The function returns a *status* (either "Success" or a message indicating that an unused internal state is not available) and, if *status* = "Success", a *state\_handle* that points to an available *internal\_state* in the array of internal states. If *status*  $\neq$  "Success", an invalid *state\_handle* is returned.

When the *uninstantiate* function is invoked in the following examples, the function specified in Section 9.5 is called.

### E.2 HMAC\_DRBG Example

#### E.2.1 Discussion

This example of **HMAC\_DRBG** uses the SHA-256 hash function. Reseeding and prediction resistance are not supported. The nonce for instantiation consists of a random value with *security\_strength/2* bits of entropy; the nonce is obtained by increasing the call for entropy bits via the **Get\_entropy\_input** call by *security\_strength/2* bits (i.e., by adding *security\_strength/2* bits to the *security\_strength* value). The **HMAC\_DRBG\_Update** function is specified in Section 10.2.2.2.2.

A personalization string is supported, but additional input is not. A total of 3 internal states are provided. For this implementation, the functions and algorithms are written as separate routines. Also, the **Get\_entropy\_input** function uses only two input parameters, since the first two parameters (as specified in Section 9) have the same value.

The internal state contains the values for *V*, *Key*, *reseed\_counter*, and *security\_strength*, where *V* and *C* are bitstrings, and *reseed\_counter* and *security\_strength* are integers.

In accordance with Table 2 in Section 10.2.1, security strengths of 112, 128, 192 and 256 bits may be instantiated. Using SHA-256, the following definitions are applicable for the *instantiate* and *generate* functions and algorithms:

1. *highest\_supported\_security\_strength* = 256.

2. Output block (*outlen*) = 256 bits.
3. Required minimum entropy for the entropy input at instantiation =  $3/2 \text{ security\_strength}$  (this includes the entropy required for the nonce).
4. Seed length (*seedlen*) = 440 bits.
5. Maximum number of bits per request (*max\_number\_of\_bits\_per\_request*) = 7500 bits.
6. Reseed\_interval (*reseed\_interval*) = 10,000 requests.
7. Maximum length of the personalization string (*max\_personalization\_string\_length*) = 160 bits.
8. Maximum length of the entropy input (*max\_length*) = 1000 bits.

### E.2.2 Instantiation of HMAC\_DRBG

This implementation will return a text message and an invalid state handle (-1) when an error is encountered.

#### HMAC\_DRBG\_Instantiate\_function:

**Input:** integer (*requested\_instantiation\_security\_strength*), bitstring *personalization\_string*.

**Output:** string *status*, integer *state\_handle*.

#### Process:

Check the validity of the input parameters.

1. If (*requested\_instantiation\_security\_strength* > 256), then **Return** ("Invalid *requested\_instantiation\_security\_strength*", -1).
2. If (**len** (*personalization\_string*) > 160), then **Return** ("*Personalization\_string* too long", -1)

Comment: Set the *security\_strength* to one of the valid security strengths.

3. If (*requested\_security\_strength* ≤ 112), then *security\_strength* = 112  
Else (*requested\_security\_strength* ≤ 128), then *security\_strength* = 128  
Else (*requested\_security\_strength* ≤ 192), then *security\_strength* = 192  
Else *security\_strength* = 256.

Comment: Get the *entropy\_input* and the *nonce*.

4. *min\_entropy* =  $1.5 \times \text{security\_strength}$ .
5. (*status*, *entropy\_input*) = **Get\_entropy\_input** (*min\_entropy*, 1000).

6. If (*status* ≠ “Success”), then **Return** (“Catastrophic failure of the entropy source.” || *status*, -1).

Comment: Invoke the instantiate algorithm.  
Note that the *entropy\_input* contains the nonce.

7. (*V*, *Key*, *reseed\_counter*) = **Instantiate\_algorithm** (*entropy\_input*, *personalization\_string*).

Comment: Find an unused internal state and save the initial values.

8. (*status*, *state\_handle*) = **Find\_state\_space** ( ).
9. If (*status* ≠ “Success”), then **Return** (“No available state space.” || *status*, -1).
10. *internal\_state* (*state\_handle*) = {*V*, *Key*, *reseed\_counter*, *security\_strength*}.
11. **Return** (“Success” and *state\_handle*).

**Instantiate\_algorithm (...):**

**Input:** bitstring (*entropy\_input*, *personalization\_string*).

**Output:** bitstring (*V*, *Key*), integer *reseed\_counter*.

**Process:**

1. *seed\_material* = *entropy\_input* || *personalization\_string*.
2. Set *Key* to *outlen* bits of zeros.
3. Set *V* to *outlen*/8 bytes of 0x01.
4. (*Key*, *V*) = **HMAC\_DRBG\_Update** (*seed\_material*, *Key*, *V*).
5. *reseed\_counter* = 1.
6. **Return** (*V*, *Key*, *reseed\_counter*).

**E.2.3 Generating Pseudorandom Bits Using HMAC\_DRBG**

The implementation returns a *Null* string as the pseudorandom bits if an error has been detected.

**HMAC\_DRBG\_Generate\_function:**

**Input:** integer (*state\_handle*, *requested\_no\_of\_bits*, *requested\_security\_strength*).

**Output:** string (*status*), bitstring *pseudorandom\_bits*.

**Process:**

Comment: Check for a valid state handle.

1. If ((*state\_handle* < 0) or (*state\_handle* > 2) or (*internal\_state* (*state\_handle*) = {*Null*, *Null*, 0, 0}), then **Return** (“State not available for the indicated *state\_handle*”, *Null*).



2. Get the internal state.
    - 2.1  $V = \text{internal\_state}(\text{state\_handle}).V$ .
    - 2.2  $\text{Key} = \text{internal\_state}(\text{state\_handle}).\text{Key}$ .
    - 2.3  $\text{security\_strength} = \text{internal\_state}(\text{state\_handle}).\text{security\_strength}$ .
    - 2.4  $\text{reseed\_counter} = \text{internal\_state}(\text{state\_handle}).\text{reseed\_counter}$ .

Comment: Check the validity of the rest of the input parameters.
  3. If ( $\text{requested\_no\_of\_bits} > 7500$ ), then **Return** (“Too many bits requested”, *Null*).
  4. If ( $\text{requested\_security\_strength} > \text{security\_strength}$ ), then **Return** (“Invalid requested\_security\_strength”, *Null*).
- Comment: Invoke the generate algorithm.
5. ( $\text{status}, \text{pseudorandom\_bits}, V, \text{Key}, \text{reseed\_counter}$ ) = **HMAC\_DRBG\_Generate\_algorithm** ( $V, \text{Key}, \text{reseed\_counter}, \text{requested\_number\_of\_bits}$ ).
  6. If ( $\text{status} = \text{“Reseed required”}$ ), then **Return** (“DRBG can no longer be used. Please re-instantiate or reseed”, *Null*).
  7. Update the changed state values.
    - 7.1  $\text{internal\_state}(\text{state\_handle}).V = V$ .
    - 7.2  $\text{internal\_state}(\text{state\_handle}).\text{Key} = \text{Key}$ .
    - 7.3  $\text{internal\_state}(\text{state\_handle}).\text{reseed\_counter} = \text{reseed\_counter}$ .
  8. **Return** (“Success”,  $\text{pseudorandom\_bits}$ ).

**HMAC\_DRBG\_Generate\_algorithm:**

**Input:** bitstring ( $V, \text{Key}$ ), integer ( $\text{reseed\_counter}, \text{requested\_number\_of\_bits}$ ).

**Output:** string  $\text{status}$ , bitstring ( $\text{pseudorandom\_bits}, V, \text{Key}$ ), integer  $\text{reseed\_counter}$ .

**Process:**

- 1 If ( $\text{reseed\_counter} \geq 10,000$ ), then **Return** (“Reseed required”, *Null*,  $V, \text{Key}, \text{reseed\_counter}$ ).
- 2  $\text{temp} = \text{Null}$ .
- 3 While ( $\text{len}(\text{temp}) < \text{requested\_no\_of\_bits}$ ) do:
  - 3.1  $V = \text{HMAC}(\text{Key}, V)$ .
  - 3.2  $\text{temp} = \text{temp} \parallel V$ .
- 4  $\text{pseudorandom\_bits} = \text{Leftmost}(\text{requested\_no\_of\_bits}) \text{ of } \text{temp}$ .

5.  $(Key, V) = \text{HMAC\_DRBG\_Update}(Null, Key, V)$ .
6.  $reseed\_counter = reseed\_counter + 1$ .
7. **Return** ("Success", *pseudorandom\_bits*, *V*, *Key*, *reseed\_counter*).

### E.3 CTR\_DRBG Example Using a Derivation Function

#### E.3.1 Discussion

This example of **CTR\_DRBG** uses AES-128. The reseed and prediction resistance capabilities are supported, and a block cipher derivation function using AES-128 is used. Both a personalization string and additional input are supported. A total of 5 internal states are available. For this implementation, the functions and algorithms are written as separate routines. **AES\_ECB\_Encrypt** is the **Block\_Encrypt** function (specified in Section 10.5.3) that uses AES-128 in the ECB mode.

The nonce for instantiation (*instantiation\_nonce*) consists of a 32-bit incrementing counter. The nonce is initialized when the DRBG is instantiated (e.g., by a call to the clock or by setting it to a fixed value) and is incremented for each instantiation.

The internal state contains the values for *V*, *Key*, *reseed\_counter*, and *security\_strength*, where *V* and *Key* are bitstrings, and all other values are integers. Since prediction resistance is known to be supported, there is no need for *prediction\_resistance\_flag* in the internal state.

In accordance with Table 3 in Section 10.3.2.1, security strengths of 112 and 128 bits may be supported. Using AES-128, the following definitions are applicable for the instantiate, reseed and generate functions:

1. *highest\_supported\_security\_strength* = 128.
2. Output block length (*outlen*) = 128 bits.
3. Key length (*keylen*) = 128 bits.
4. Required minimum entropy for the entropy input during instantiation and reseeding = *security\_strength*.
5. Minimum entropy input length (*min\_length*) = *security\_strength* bits.
6. Maximum entropy input length (*max\_length*) = 1000 bits.
7. Maximum personalization string input length (*max\_personalization\_string\_input\_length*) = 800 bits.
8. Maximum additional input length (*max\_additional\_input\_length*) = 800 bits.
9. Seed length (*seedlen*) = 256 bits.
10. Maximum number of bits per request (*max\_number\_of\_bits\_per\_request*) = 4000 bits.
11. Reseed interval (*reseed\_interval*) = 100,000 requests.

#### E.3.2 The CTR\_DRBG\_Update Function

##### CTR\_DRBG\_Update:

**Input:** bitstring (*provided\_data*, *Key*, *V*).

**Output:** bitstring (*Key*, *V*).

**Process:**

1. *temp* = Null.
2. While (**len** (*temp*) < 256) do
  - 3.1  $V = (V + 1) \bmod 2^{128}$ .
  - 3.2 *output\_block* = **AES\_ECB\_Encrypt** (*Key*, *V*).
  - 3.3 *temp* = *temp* || *output\_block*.
4. *temp* = Leftmost 256 bits of *temp*.
5. *temp* = *temp*  $\oplus$  *provided\_data*.
6. *Key* = Leftmost 128 bits of *temp*.
7. *V* = Rightmost 128 bits of *temp*.
8. **Return** (*Key*, *V*).

### E.3.3 Instantiation of CTR\_DRBG Using a Derivation Function

This implementation will return a text message and an invalid state handle (-1) when an error is encountered. **Block\_Cipher\_df** is the derivation function in Section 10.5.3, and uses AES-128 in the ECB mode as the **Block\_Encrypt** function.

Note that this implementation does not include the *prediction\_resistance\_flag* in the input parameters, nor save it in the internal state, since prediction resistance is known to be supported.

**CTR\_DRBG\_Instantiate function:**

**Input:** integer (*requested\_instantiation\_security\_strength*), bitstring *personalization\_string*.

**Output:** string *status*, integer *state\_handle*.

**Process:**

Comment: Check the validity of the input parameters.

1. If (*requested\_instantiation\_security\_strength* > 128) then **Return** ("Invalid *requested\_instantiation\_security\_strength*", -1).
2. If (**len** (*personalization\_string*) > 800), then **Return** ("*Personalization\_string* too long", -1).
3. If (*requested\_instantiation\_security\_strength*  $\leq$  112), then *security\_strength* = 112

Else *security\_strength* = 128.

Comment: Get the entropy input.

4. (*status*, *entropy\_input*) = **Get\_entropy\_input** (*security\_strength*, *security\_strength*, 1000).
5. If (*status* ≠ "Success"), then **Return** ("Catastrophic failure of the entropy source" || *status*, -1).

Comment: Increment the nonce; actual coding must ensure that the nonce wraps when its storage limit is reached, and that the counter pertains to all instantiations, not just this one.

6. *instantiation\_nonce* = *instantiation\_nonce* + 1.

Comment: Invoke the instantiate algorithm.

7. (*V*, *Key*, *reseed\_counter*) = **CTR\_DRBG\_Instantiate\_algorithm** (*entropy\_input*, *instantiation\_nonce*, *personalization\_string*).

Comment: Find an available internal state and save the initial values.

8. (*status*, *state\_handle*) = **Find\_state\_space** ( ).
9. If (*status* ≠ "Success"), then **Return** ("No available state space:" || *status*, -1).

Comment: Save the internal state.

10. Save the internal state.

10.1 *internal\_state\_*(*state\_handle*).*V* = *V*.

10.2 *internal\_state\_*(*state\_handle*).*Key* = *Key*.

10.3 *internal\_state\_*(*state\_handle*).*reseed\_counter* = *reseed\_counter*.

10.4 *internal\_state\_*(*state\_handle*).*security\_strength* = *security\_strength*.

11. **Return** ("Success", *state\_handle*).

#### **CTR\_DRBG\_Instantiate\_algorithm:**

**Input:** bitstring (*entropy\_input*, *nonce*, *personalization\_string*).

**Output:** bitstring (*V*, *Key*), integer (*reseed\_counter*).

**Process:**

1. *seed\_material* = *entropy\_input* || *nonce* || *personalization\_string*.
2. *seed\_material* = **Block\_Cipher\_df** (*seed\_material*, 256).
3. *Key* =  $0^{128}$ . Comment: 128 bits.
4. *V* =  $0^{128}$ . Comment: 128 bits.

5.  $(Key, V) = \text{CTR\_DRBG\_Update}(\text{seed\_material}, Key, V)$ .
6.  $\text{reseed\_counter} = 1$ .
7. **Return**  $(V, Key, \text{reseed\_counter})$ .

#### E.3.4 Reseeding a CTR\_DRBG Instantiation Using a Derivation Function

The implementation is designed to return a text message as the *status* when an error is encountered.

##### CTR\_DRBG\_Reseed\_function:

**Input:** integer (*state\_handle*), bitstring *additional\_input*.

**Output:** string *status*.

**Process:**

Comment: Check for the validity of  
*state\_handle*.

1. If  $((\text{state\_handle} < 0) \text{ or } (\text{state\_handle} > 4) \text{ or } (\text{internal\_state}(\text{state\_handle}) = \{\text{Null}, \text{Null}, 0, 0\}))$ , then **Return** ("State not available for the indicated *state\_handle*").
2. Get the internal state values.
  - 2.1  $V = \text{internal\_state}(\text{state\_handle}).V$ .
  - 2.2  $Key = \text{internal\_state}(\text{state\_handle}).Key$ .
  - 2.3  $\text{security\_strength} = \text{internal\_state}(\text{state\_handle}).\text{security\_strength}$ .
3. If  $(\text{len}(\text{additional\_input}) > 800)$ , then **Return** ("additional\_input too long").
4.  $(\text{status}, \text{entropy\_input}) = \text{Get\_entropy\_input}(\text{security\_strength}, \text{security\_strength}, 1000)$ .
6. If  $(\text{status} \neq \text{"Success"})$ , then **Return** ("Catastrophic failure of the entropy source:" || *status*).

Comment: Invoke the reseed algorithm.

7.  $(V, Key, \text{reseed\_counter}) = \text{CTR\_DRBG\_Reseed\_algorithm}(V, Key, \text{reseed\_counter}, \text{entropy\_input}, \text{additional\_input})$ .
8. Save the internal state:
  - 8.1  $\text{internal\_state}(\text{state\_handle}).V = V$ .
  - 8.2  $\text{internal\_state}(\text{state\_handle}).Key = Key$ .
  - 8.3  $\text{internal\_state}(\text{state\_handle}).\text{reseed\_counter} = \text{reseed\_counter}$ .
  - 8.4  $\text{internal\_state}(\text{state\_handle}).\text{security\_strength} = \text{security\_strength}$ .
9. **Return** ("Success").

**CTR\_DRBG\_Reseed\_algorithm (...):**

**Input:** bitstring ( $V$ ,  $Key$ ), integer ( $reseed\_counter$ ), bitstring ( $entropy\_input$ ,  $additional\_input$ ).

**Output:** bitstring ( $V$ ,  $Key$ ), integer ( $reseed\_counter$ ).

**Process:**

1.  $seed\_material = entropy\_input \parallel additional\_input$ .
2.  $seed\_material = \text{Block\_Cipher\_df}(seed\_material, 256)$ .
3.  $(Key, V) = \text{CTR\_DRBG\_Update}(seed\_material, Key, V)$ .
4.  $reseed\_counter = 1$ .
5. **Return**  $V, Key, reseed\_counter$ .

**E.3.5 Generating Pseudorandom Bits Using CTR\_DRBG**

The implementation returns a *Null* string as the pseudorandom bits if an error has been detected.

**CTR\_DRBG\_Generate\_function:**

**Input:** integer ( $state\_handle$ ,  $requested\_no\_of\_bits$ ,  $requested\_security\_strength$ ,  $prediction\_resistance\_request$ ), bitstring  $additional\_input$ .

**Output:** string  $status$ , bitstring  $pseudorandom\_bits$ .

**Process:**

Comment: Check the validity of  $state\_handle$ .

1. If  $((state\_handle < 0) \text{ or } (state\_handle > 4) \text{ or } (internal\_state(state\_handle) = \{Null, Null, 0, 0\}))$ , then **Return** ("State not available for the indicated  $state\_handle$ ", *Null*).
2. Get the internal state values.
  - 2.1  $V = internal\_state(state\_handle).V$ .
  - 2.2  $Key = internal\_state(state\_handle).Key$ .
  - 2.3  $security\_strength = internal\_state(state\_handle).security\_strength$ .
  - 2.4  $reseed\_counter = internal\_state(state\_handle).reseed\_counter$ .

Comment: Check the rest of the input parameters.

3. If  $(requested\_no\_of\_bits > 4000)$ , then **Return** ("Too many bits requested", *Null*).
4. If  $(requested\_security\_strength > security\_strength)$ , then **Return** ("Invalid  $requested\_security\_strength$ ", *Null*).

5. If (**len** (*additional\_input*) > 800), then **Return** (“*additional\_input* too long”, *Null*).
6. *reseed\_required\_flag* = 0.
7. If ((*reseed\_required\_flag* = 1) OR (*prediction\_resistance\_flag* = 1)), then
  - 7.1 *status* = **CTR\_DRBG\_Reseed\_function** (*state\_handle*, *additional\_input*).
  - 7.2 If (*status* ≠ “Success”), then **Return** (*status*, *Null*).
  - 7.3 Get the new working state values.
    - 7.3.1 *V* = *internal\_state* (*state\_handle*).*V*.
    - 7.3.2 *Key* = *internal\_state* (*state\_handle*).*Key*.
    - 7.3.3 *reseed\_counter* = *internal\_state* (*state\_handle*).*reseed\_counter*.
  - 7.4 *additional\_input* = *Null*.
  - 7.5 *reseed\_required\_flag* = 0.

Comment: Generate bits using the generate algorithm.

8. (*status*, *pseudorandom\_bits*, *V*, *Key*, *reseed\_counter*) = **CTR\_DRBG\_Generate\_algorithm** (*V*, *Key*, *reseed\_counter*, *requested\_number\_of\_bits*, *additional\_input*).
9. If (*status* = “Reseed required”), then
  - 9.1 *reseed\_required\_flag* = 1.
  - 9.2 Go to step 7.
10. Update the internal state.
  - 10.1 *internal\_state* (*state\_handle*).*V* = *V*.
  - 10.2 *internal\_state* (*state\_handle*).*Key* = *Key*.
  - 10.3 *internal\_state* (*state\_handle*). *reseed\_counter* = *reseed\_counter*.
  - 10.4 *internal\_state* (*state\_handle*). *security\_strength* = *security\_strength*.
11. **Return** (“Success”, *pseudorandom\_bits*).

**CTR\_DRBG\_Generate\_algorithm (...):**

**Input:** bitstring (*V*, *Key*), integer (*reseed\_counter*, *requested\_number\_of\_bits*)  
bitstring *additional\_input*.

**Output:** string *status*, bitstring (*returned\_bits*, *V*, *Key*), integer *reseed\_counter*.

**Process:**

1. If (*reseed\_counter* > 100,000), then **Return** ("Reseed required", *Null*, *V*, *Key*, *reseed\_counter*).
2. If (*additional\_input* ≠ *Null*), then
  - 2.1 *additional\_input* = **Block\_Cipher\_df** (*additional\_input*, 256).
  - 2.2 (*Key*, *V*) = **CTR\_DRBG\_Update** (*additional\_input*, *Key*, *V*).Else *additional\_input* = 0<sup>256</sup>.
3. *temp* = *Null*.
4. While (**len** (*temp*) < *requested\_number\_of\_bits*) do:
  - 4.1 *V* = (*V* + 1) mod 2<sup>128</sup>.
  - 4.2 *output\_block* = **AES\_ECB\_Encrypt** (*Key*, *V*).
  - 4.3 *temp* = *temp* || *output\_block*.
5. *returned\_bits* = Leftmost (*requested\_number\_of\_bits*) of *temp*.
6. (*Key*, *V*) = **CTR\_DRBG\_Update** (*additional\_input*, *Key*, *V*)
- 7 *reseed\_counter* = *reseed\_counter* + 1.
8. **Return** ("Success", *returned\_bits*, *V*, *Key*, *reseed\_counter*).

#### E.4 CTR\_DRBG Example Without a Derivation Function

##### E.4.1 Discussion

This example of **CTR\_DRBG** is the same as the previous example except that a derivation function is not used. As in Annex E.3, the **CTR\_DRBG** uses AES-128. The reseed and prediction resistance capabilities are available. Both a personalization string and additional input are supported. A total of 5 internal states are available. For this implementation, the functions and algorithms are written as separate routines. **AES\_ECB\_Encrypt** is the **Block\_Encrypt** function (as specified in Section 10.5.4) that uses AES-128 in the ECB mode.

The nonce for instantiation (*instantiation\_nonce*) consists of a 32-bit incrementing counter that is the leftmost bits of the personalization string (Section 8.5.2 states that when a derivation function is used, the nonce, if used, is contained in the personalization string). The nonce is initialized when the DRBG is instantiated (e.g., by a call to the clock or by setting it to a fixed value) and is incremented for each instantiation.

The internal state contains the values for *V*, *Key*, *reseed\_counter*, and *security\_strength*, where *V* and *Key* are strings, and all other values are integers. Since prediction resistance is known to be supported, there is no need for *prediction\_resistance\_flag* in the internal state.

In accordance with Table 3 in Section 10.3.2.1, security strengths of 112 and 128 bits may be supported. The definitions are the same as those provided in Annex E.3, except that to be compliant with Table 3, the maximum size of the *personalization\_string* is 224 bits in order to accommodate the 32-bits of the *instantiation\_nonce* (i.e., **len** (*instantiation\_nonce*) + **len**



(*personalization\_string*) must be  $\leq seedlen$ , where  $seedlen = 256$  bits). In addition, the maximum size of any *additional\_input* is 256 bits (i.e.,  $len(additional\_input) \leq seedlen$ )).

#### E.4.2 The CTR\_DRBG\_Update Function

The update function is the same as that provided in Annex E.3.2.

#### E.4.3 Instantiation of CTR\_DRBG Without a Derivation Function

The instantiate function (**CTR\_DRBG\_Instantiate\_function**) is the same as that provided in Annex E.3.3, except for the following:

- Step 2 is replaced by:  
If ( $len(personalization\_string) > 224$ ), then **Return** ("Personalization\_string too long", -1).
- Step 6 is replaced by :  
 $instantiation\_nonce = instantiation\_nonce + 1$ .  
 $personalization\_string = instantiation\_nonce \parallel personalization\_string$ .

The instantiate algorithm (**CTR\_DRBG\_Instantiate\_algorithm**) is the same as that provided in Annex E.3.3, except that steps 1 and 2 are replaced by:

$temp = len(personalization\_string)$ .  
If ( $temp < 256$ ), then  $personalization\_string = personalization\_string \parallel 0^{256-temp}$ .  
 $seed\_material = entropy\_input \oplus personalization\_string$ .

#### E.4.4 Reseeding a CTR\_DRBG Instantiation Without a Derivation Function

The reseed function (**CTR\_DRBG\_Reseed\_function**) is the same as that provided in Annex E.3.4, except that step 3 is replaced by:

If ( $len(additional\_input) > 256$ ), then **Return** ("additional\_input too long").

The reseed algorithm (**CTR\_DRBG\_Reseed\_algorithm**) is the same as that provided in Annex E.3.4, except that steps 1 and 2 are replaced by:

$temp = len(additional\_input)$ .  
If ( $temp < 256$ ), then  $additional\_input = additional\_input \parallel 0^{256-temp}$ .  
 $seed\_material = entropy\_input \oplus additional\_input$ .

#### E.4.5 Generating Pseudorandom Bits Using CTR\_DRBG

The generate function (**CTR\_DRBG\_Generate\_function**) is the same as that provided in Annex E.3.5, except that step 5 is replaced by :

If ( $len(additional\_input) > 256$ ), then **Return** ("additional\_input too long", *Null*).

The generate algorithm (**CTR\_DRBG\_Generate\_algorithm**) is the same as that provided in Annex E.3.5, except that step 2.1 is replaced by:

$temp = \text{len}(\text{additional\_input})$ .

If ( $temp < 256$ ), then  $\text{additional\_input} = \text{additional\_input} \parallel 0^{256-temp}$ .

## E.5 Dual\_EC\_DRBG Example

### E.5.1 Discussion

This example of **Dual\_EC\_DRBG** allows a consuming application to instantiate using any of the three prime curves. The elliptic curve to be used is selected during instantiation in accordance with the following:

<i>requested_instantiation_security_strength</i>	Elliptic Curve
$\leq 112$	P-256
113 – 128	P-256
129 – 192	P-384
193 – 256	P-521

A reseed capability is available, but prediction resistance is supported. Both a *personalization\_string* and an *additional\_input* are allowed. A total of 10 internal states are provided. For this implementation, the algorithms are provided as inline code within the functions.

The nonce for instantiation (*instantiation\_nonce*) consists of a random value with  $\text{security\_strength}/2$  bits of entropy; the nonce is obtained by a separate call to the **Get\_entropy\_input** routine than that used to obtain the entropy input itself. Also, the **Get\_entropy\_input** function uses only two input parameters, since the first two parameters (the *min\_entropy* and the *min\_length*) have the same value.

The internal state contains values for *s*, *seedlen*, *p*, *a*, *b*, *n*, *P*, *Q*, *reseed\_counter* and *security\_strength*.

In accordance with Table 4 in Section 10.4.2, security strengths of 112, 128, 192 and 256 bits may be supported. SHA-256 has been selected as the hash function. The following definitions are applicable for the instantiate, reseed and generate functions:

1.  $\text{highest\_supported\_security\_strength} = 256$ .
2. Output block length (*outlen*) = *max\_outlen*. See Table 4.
3. Required minimum entropy for the entropy input at instantiation and reseed = *security\_strength*.
4. Maximum entropy input length (*max\_length*) = 1000 bits.
5. Maximum personalization string length (*max\_personalization\_string\_length*) = 800 bits.
6. Maximum additional input length (*max\_additional\_input\_length*) = 800 bits.

7. Seed length (*seedlen*):  $= 2 \times \text{security\_strength}$ .
8. Maximum number of bits per request (*max\_number\_of\_bits\_per\_request*) = 1000 bits.
9. Reseed interval (*reseed\_interval*) =  $2^{32}$  blocks.

#### E.5.2 Instantiation of Dual\_EC\_DRBG

This implementation will return a text message and an invalid state handle (-1) when an **ERROR** is encountered. **Hash\_df** is specified in Section 10.5.2.

##### Dual\_EC\_DRBG\_Instantiate\_function:

**Input:** integer (*requested\_instantiation\_security\_strength*), bitstring *personalization\_string*.

**Output:** string *status*, integer *state\_handle*.

##### Process:

Comment : Check the validity of the input parameters.

1. If (*requested\_instantiation\_security\_strength* > 256) then **Return** ("Invalid *requested\_instantiation\_security\_strength*", -1).
2. If (**len** (*personalization\_string*) > 800), then **Return** ("*personalization\_string* too long", -1).

Comment : Select the prime field curve in accordance with the *requested\_instantiation\_security\_strength*.

3. If *requested\_instantiation\_security\_strength* ≤ 112), then  
    {*security\_strength* = 112; *seedlen* = 224; *outlen* = 240}  
    Else if (*requested\_instantiation\_security\_strength* ≤ 128), then  
        {*security\_strength* = 128; *seedlen* = 256; *outlen* = 240}  
    Else if (*requested\_instantiation\_security\_strength* ≤ 192), then  
        {*security\_strength* = 192; *seedlen* = 384; *outlen* = 368}  
    Else {*security\_strength* = 256; *seedlen* = 512; *outlen* = 504}.
4. Select the appropriate elliptic curve from Annex A using the Table in Annex F.5.1 to obtain the domain parameters *p*, *a*, *b*, *n*, *P*, and *Q*.

Comment: Request *entropy\_input*.

5. (*status*, *entropy\_input*) = **Get\_entropy\_input** (*security\_strength*, 1000).

6. If (*status* ≠ "Success"), then **Return** ("Catastrophic failure of the *entropy\_input* source:" || *status*, -1).
7. (*status*, *instantiation\_nonce*) = **Get\_entropy\_input** (*security\_strength*/2, 1000).
8. If (*status* ≠ "Success"), then **Return** ("Catastrophic failure of the random nonce source:" || *status*, -1).

Comment: Perform the instantiate algorithm.

9. *seed\_material* = *entropy\_input* || *instantiation\_nonce* || *personalization\_string*.
10. *s* = **Hash\_df** (*seed\_material*, *seedlen*).
11. *reseed\_counter* = 0.

Comment: Find an unused internal state and save the initial values.

12. (*status*, *state\_handle*) = **Find\_state\_space** ( ).
13. If (*status* ≠ "Success"), then **Return** (*status*, -1).
14. Save the internal state.
  - 14.1 *internal\_state* (*state\_handle*).*s* = *s*.
  - 14.2 *internal\_state* (*state\_handle*).*seedlen* = *seedlen*.
  - 14.3 *internal\_state* (*state\_handle*).*p* = *p*.
  - 14.4 *internal\_state* (*state\_handle*).*a* = *a*.
  - 14.5 *internal\_state* (*state\_handle*).*b* = *b*.
  - 14.6 *internal\_state* (*state\_handle*).*n* = *n*.
  - 14.7 *internal\_state* (*state\_handle*).*P* = *P*.
  - 14.8 *internal\_state* (*state\_handle*).*Q* = *Q*.
  - 14.9 *internal\_state* (*state\_handle*).*reseed\_counter* = *reseed\_counter*.
  - 14.10 *internal\_state* (*state\_handle*).*security\_strength* = *security\_strength*.
15. **Return** ("Success", *state\_handle*).

### E.5.3 Reseeding a Dual\_EC\_DRBG Instantiation

The implementation is designed to return a text message as the status when an error is encountered.

#### Dual\_EC\_DRBG\_Reseed\_function:

**Input:** integer *state\_handle*, string *additional\_input*.

**Output:** string *status*.

**Process:**

Comment: Check the input parameters.

1. If  $((state\_handle < 0) \text{ or } (state\_handle > 9) \text{ or } (internal\_state(state\_handle).security\_strength = 0))$ , then **Return** ("State not available for the *state\_handle*").
2. If  $(len(additional\_input) > 800)$ , then **Return** ("*additional\_input* too long").
3. Get the appropriate *state* values for the indicated *state\_handle*.
  - 3.1  $s = internal\_state(state\_handle).s$ .
  - 3.2  $seedlen = internal\_state(state\_handle).seedlen$ .
  - 3.3  $security\_strength = internal\_state(state\_handle).security\_strength$ .

Comment: Request new *entropy\_input* with the appropriate entropy and bit length.

4.  $(status, entropy\_input) = \text{Get\_entropy\_input}(security\_strength, 1000)$ .
5. If  $(status \neq \text{"Success"})$ , then **Return** ("Catastrophic failure of the entropy source:" || *status*).

Comment: Perform the reseed algorithm.

6.  $seed\_material = \text{pad8}(s) \parallel entropy\_input \parallel additional\_input$ .
7.  $s = \text{Hash\_df}(seed\_material, seedlen)$ .
8. Update the changed values in the *state*.
  - 8.1  $internal\_state(state\_handle).s = s$ .
  - 8.2  $internal\_state.reseed\_counter = 0$ .
9. **Return** ("Success").

#### E.5.4 Generating Pseudorandom Bits Using Dual\_EC\_DRBG

The implementation returns a *Null* string as the pseudorandom bits if an error is encountered.

##### Dual\_EC\_DRBG\_Generate\_function:

**Input:** integer (*state\_handle*, *requested\_security\_strength*, *requested\_no\_of\_bits*),  
bitstring *additional\_input*.

**Output:** string *status*, bitstring *pseudorandom\_bits*.

**Process:**

Comment: Check for an invalid *state\_handle*.

1. If  $((state\_handle < 0) \text{ or } (state\_handle > 9) \text{ or } (internal\_state(state\_handle) = 0))$ , then **Return** ("State not available for the *state\_handle*", *Null*).

2. Get the appropriate *state* values for the indicated *state\_handle*.
  - 2.1  $s = \text{internal\_state}(\text{state\_handle}).s$ .
  - 2.2  $\text{seedlen} = \text{internal\_state}(\text{state\_handle}).\text{seedlen}$ .
  - 2.3  $P = \text{internal\_state}(\text{state\_handle}).P$ .
  - 2.4  $Q = \text{internal\_state}(\text{state\_handle}).Q$ .
  - 2.5  $\text{security\_strength} = \text{internal\_state}(\text{state\_handle}).\text{security\_strength}$ .
  - 2.6  $\text{reseed\_counter} = \text{internal\_state}(\text{state\_handle}).\text{reseed\_counter}$ .

Comment: Check the rest of the input parameters.
3. If  $(\text{requested\_number\_of\_bits} > 1000)$ , then **Return** (“Too many bits requested”, *Null*).
4. If  $(\text{requested\_security\_strength} > \text{security\_strength})$ , then **Return** (“Invalid requested\_strength”, *Null*).
5. If  $(\text{len}(\text{additional\_input}) > 800)$ , then **Return** (“additional\_input too long”, *Null*).

Comment: Check whether a reseed is required.

6. If  $(\text{reseed\_counter} + \left\lceil \frac{\text{requested\_number\_of\_bits}}{\text{outlen}} \right\rceil > 2^{32})$ , then
  - 6.1 **Dual\_EC\_DRBG\_Reseed\_function** (*state\_handle*, *additional\_input*).
  - 6.2 If  $(\text{status} \neq \text{“Success”})$ , then **Return** (*status*).
  - 6.3  $s = \text{internal\_state}(\text{state\_handle}).s$ ,  $\text{reseed\_counter} = \text{internal\_state}(\text{state\_handle}).\text{reseed\_counter}$ .
  - 6.4  $\text{additional\_input} = \text{Null}$ .

Comment: Execute the generate algorithm.

7. If  $(\text{additional\_input} = \text{Null})$  then  $\text{additional\_input} = 0$

Comment: *additional\_input* set to *m* zeroes.

Else  $\text{additional\_input} = \text{Hash\_df}(\text{pad8}(\text{additional\_input}), \text{seedlen})$ .

Comment: Produce *requested\_no\_of\_bits*,  
*outlen* bits at a time:

8.  $\text{temp} = \text{the Null string}$ .

9.  $i = 0$ .
10.  $t = s \oplus \text{additional\_input}$ .
11.  $s = \varphi(x(t * P))$ .
12.  $r = \varphi(x(s * Q))$ .
13.  $\text{temp} = \text{temp} \parallel (\text{rightmost outlen bits of } r)$ .
14.  $\text{additional\_input} = 0^{\text{seedlen}}$ .      Comment: *seedlen* zeroes; *additional\_input* is added only on the first iteration.
15.  $\text{reseed\_counter} = \text{reseed\_counter} + 1$ .
16.  $i = i + 1$ .
17. If (**len** (*temp*) < *requested\_no\_of\_bits*), then go to step 10.
18.  $\text{pseudorandom\_bits} = \text{Truncate}(\text{temp}, i \times \text{outlen}, \text{requested\_no\_of\_bits})$ .
19. Update the changed values in the *state*.
  - 19.1  $\text{internal\_state}.s = \varphi(x(s * P))$ .
  - 19.2  $\text{internal\_state}. \text{reseed\_counter} = \text{reseed\_counter}$ .
20. **Return** ("Success", *pseudorandom\_bits*).

## **ANNEX F: (Informative) DRBG Provision of RBG Security Properties**

### **F.1 Introduction**

Part 1 of this Standard identifies several security properties that are required for cryptographic random number generators. This annex discusses how these properties are provided by the DRBG mechanisms in this part of the Standard or points to sections in Part 3 or in other parts of the Standard that will provide appropriate guidance for fulfilling the security properties.

### **F.2 Security Strengths**

Part 1 identifies four security strengths that RBGs support: 112, 128, 192 and 256 bits. These security strengths may be supported in Part 3 by requesting the appropriate security strength during instantiation and generation (see Sections 8.2.4, 9.2 and 9.4), and by the use of an appropriate source of entropy input (see Part 4).

### **F.3 Entropy and Min-Entropy**

Part 1 defines the use of min-entropy to measure the amount of entropy needed to support a given security strength. Part 3 requests the entropy via the use of a **Get\_entropy\_input** call (see Section 9.1). Part 4 provide guidance on supporting this call.

### **F.4 Backtracking Resistance and Prediction Resistance**

Part 1 defines backtracking and prediction resistance. As indicated in Section 8.6, the DRBG mechanisms in Part 3 have been designed to support backtracking resistance. Prediction resistance may be provided using a DRBG when:

1. A reseed capability is available that can obtain the appropriate amount of entropy required to support the security strength of the instantiation during each call for entropy input (see Section 9.3),
2. A prediction resistance flag that is used as input during instantiation indicates that prediction resistance may be required for the instantiation (see Section 9.2), and
3. A prediction resistance request is made in a generate request (see Section 9.4).

### **F.5 Indistinguishability and Unpredictability**

Part 1 states that this Standard requires indistinguishability from random, in addition to unpredictability for RBG output. The DRBG mechanisms in this Standard have been designed to provide these properties when provided with sufficient entropy as discussed in Part 4.

### **F.6 Desired RBG Output Properties**

Part 1 states that the output of a cryptographically secure RBG has the following desired properties:

1. Under reasonable assumptions, it is not feasible to distinguish the output of the RBG from true random numbers that are uniformly distributed with or without replacement.



Informally, all possible outputs occur with equal probability, and a series of outputs appears to conform to a uniform distribution.

2. Given only a sequence of output bits, it is not feasible to compute or predict any other output bit, either past or future. Note that this is different from both prediction resistance and backtracking resistance.
3. The outputs of an RBG are statistically unique. That is, the output values either (A) are allowed to repeat with a negligible probability or (B) are prohibited from repeating (whether by being selected without replacement or by discarding duplicates) to meet application requirements for a specified class of outputs. Note that option B will impose constraints on the minimum output size and maximum cryptoperiod.

The DRBG mechanisms in this Standard have been designed to provide these properties when provided with sufficient entropy as discussed in Parts 2 and 4.

### **F.7 Desired RBG Operational Properties**

The desired operational properties of an RBG are as follows:

1. *The RBG does not generate bits unless the generator has been assessed to possess sufficient entropy.*

The **Get\_entropy\_input** call (see Section 9.1) is used during instantiation to obtain sufficient entropy to support the desired security strength. This property is supported if:

- a. The source of entropy input is designed and implemented as required in Parts 2 and 4 of this Standard,
  - b. Entropy input is not returned during instantiation unless the requested amount of entropy has been obtained (see Section 9.2).
2. *When an error is detected, the RBG either (a) enters a permanent error state, or (b) is able to recover from a loss or compromise of entropy if the permanent error state is deemed unacceptable for the application requirements.*

Part 3 specifies the conditions that must be tested for each DRBG mechanism function (see Sections 9.2, 9.3 and 9.4), the tests to be made during health testing (see Section 11.4) and the handling of any errors detected (see Section 9.7).

3. *The design and implementation of an RBG has a defined logical protection boundary. The RBG needs to be protected in a manner that is consistent with the use and sensitivity of the output for the consuming application.*

Part 3 uses a conceptual DRBG mechanism boundary to provide this property. Requirements for the DRBG mechanism boundary are provided in Section 8.3.

4. *The probability that the RBG can "misbehave" in some pathological way that violates the output requirements (e.g., constant output or small cycles; that is, looping such that the same output is repeated) is sufficiently small.*

Assurance of this property may be obtained when an RBG implementation is validated as discussed in Sections 2 and 11.3 of Part 3, and in Parts 2 and 4.

5. *The RBG design includes methods to prohibit predictable influence, manipulation, or side-channel observation as appropriate, depending on the threat model.*

Assurance of this property may be obtained when an RBG implementation is validated as discussed in Sections 2 and 11.3 of Part 3, and in Parts 2 and 4.

6. *The RBG output does not directly leak secret information to an adversary observer.*

Assurance of this property may be obtained when an RBG implementation is validated for as discussed in Section 2 and 11.3 of Part 3, and in Parts 2 and 4.

7. *The RBG can be run in known-answer test mode. All portions that can have known-answer tests are tested in this mode. When an RBG is in known-answer test mode, the RBG is not capable of being used to generate output bits and does not use any stored secret information; however, it may use non-secret information for testing purposes.*

The health testing of a DRBG mechanism is discussed in Sections 9.6 and 11.4.

8. *An RBG is designed to support backtracking resistance.*

The DRBG mechanisms in Part 3 have been designed to support backtracking resistance (see Section 8.6).

9. *An RBG may support prediction resistance.*

A DRBG mechanism may be designed and implemented to support prediction resistance. See Annex F.4 for additional information.

## ANNEX G: (Informative) DRBG Mechanism Security Properties

### G.1 Overview

The security properties of the three DRBG mechanisms specified in this Standard may be used to specify the assumptions that may be made by the developers of consuming applications. A maximum number of output bits per Generate call, and a maximum number of Generate calls have been specified for each DRBG mechanism. These numbers allow a specification of the expected level of resistance to attacks that involve collecting large numbers of outputs and searching for internal collisions or the lack of collisions in the output.

### G.2 HMAC\_DRBG

**HMAC\_DRBG**, with an  $n$ -bit MAC, a maximum of  $2^{48}$  Generate requests, and a maximum of  $2^{16} \times n$  bits of output for each request, cannot be distinguished from random substantially more easily than guessing an  $n$ -bit secret key.

Using an Approved  $n$ -bit hash function in the HMAC construction, **HMAC\_DRBG** provides  $n$  bits of security. The DRBG mechanism generates up to  $2^{48}$  sequences of outputs, each of up to  $2^{16}$   $n$ -bit HMAC outputs. This provides a pool of  $2^{64}$  output values. Distinguishing this set of  $2^{64}$  output values from an ideal random sequence is no easier than guessing an  $n$ -bit HMAC key.

The best known distinguisher for this DRBG mechanism relies on the fact that each Generate request uses HMAC with a single key in OFB-mode to generate up to  $2^{16}$  blocks of output. In each  $2^{16}$ -block output sequence, the probability of an internal collision (a repeated output value) is approximately  $2^{31-n}$ . After  $2^{48}$  such output sequences, the probability that of an internal collision is about  $2^{79-n}$ . For SHA-1,  $n = 160$ , there is about a  $2^{-81}$  probability of such a collision. If a collision occurs, the outputs will repeat after the collision, making distinguishing the outputs from random very easy. Internal collisions between different Generate requests are not generally an issue, since each Generate request uses a new HMAC key. The probability of any two keys colliding when  $n = 160$  is about  $2^{-65}$ .

### G.3 CTR\_DRBG

Two block cipher algorithms are currently Approved for use with **CTR\_DRBG**: AES and three-key TDEA.

- **CTR\_DRBG** using AES, with a  $k$ -bit key, a maximum of  $2^{48}$  generate requests and a maximum of  $2^{19}$  bits of output for each request, cannot be distinguished from random substantially more easily than guessing a  $k$ -bit AES key.
- **CTR\_DRBG** using three-key TDEA, with a maximum of  $2^{32}$  generate requests and a maximum of  $2^{13}$  bits of output for each request, cannot be distinguished from random substantially more easily than guessing a 112-bit random key<sup>4</sup>.

**CTR\_DRBG** using an  $n$ -bit block cipher with a  $k$ -bit key and an  $s$ -bit security strength, provides  $s$  bits of security.

---

<sup>4</sup> Three-key TDEA uses a 192-bit key; this is approximately equivalent to a 112-bit random key because of shortcut attacks on three-key TDEA.

- Using AES, **CTR\_DRBG** has a 128-bit block and a key length (equivalently, a maximum security strength) of 128, 192, or 256 bits. The DRBG mechanism generates up to  $2^{48}$  sequences of outputs of at most  $2^{12}$  128-bit blocks per sequence. This provides a total of  $2^{64}$  128-bit outputs that may be generated from a single DRBG instantiation. Distinguishing the full sequence of outputs from an ideal random sequence is not significantly easier than guessing the  $k$ -bit AES key.
- Using three-key TDEA, **CTR\_DRBG** has a 64-bit block, a key length of 168 bits, and a security strength of 112 bits. The DRBG mechanism generates up to  $2^{32}$  output sequences of at most  $2^7$  64-bit blocks per sequence. Distinguishing the full sequence of outputs from an ideal random sequence is not significantly easier than guessing a 112-bit key.

The best known distinguishing attacks that do not use any properties of the block ciphers are based on the fact that in counter mode, an  $n$ -bit block never repeats within the same Generate call. This provides a property by which an ideal random sequence might be distinguished from a **CTR\_DRBG** sequence.

- AES: Consider a random sequence of  $2^{12}$  128-bit values (i.e.,  $2^{12}$  blocks, each block consisting of 128 bits). The probability of a collision (i.e., a pair of 128-bit blocks that is repeated) is about  $2^{23-128} = 2^{-105}$ . After  $2^{48}$  such values, the probability is about  $2^{-57}$  that a sequence of  $2^{48}$  blocks, each consisting of  $2^{12}$  128-bit values, would contain at least one collision. This provides a distinguisher with an advantage of  $2^{-58}$  against **CTR\_DRBG** with AES.
- Three-key TDEA: Consider a random sequence of  $2^7$  64-bit blocks (i.e.,  $2^7$  blocks, each block consisting of 64 bits). The probability of a collision (i.e., a pair of 64-bit blocks that is repeated) is about  $2^{13-64} = 2^{-51}$ . After  $2^{32}$  such outputs, the probability is about  $2^{-19}$  that a random sequence of  $2^{32}$  blocks of  $2^7$  64-bit values each would include at least one such collision. This provides a distinguisher with an advantage of  $2^{-20}$  against **CTR\_DRBG** with TDEA.

In both cases, consider the possibility of cycling. Each generate call uses a new key. Assuming that each key is random, the probability of a collision on the key for **CTR\_DRBG** with AES is expected to be about  $2^{-33}$  for 128-bit keys,  $2^{-97}$  for 192-bit keys, and  $2^{-129}$  for 256-bit keys. For three-key TDEA, the probability of a key collision in any of the  $2^{32}$  Generate calls allowed for a single DRBG instance is about  $2^{-114}$ .

If two Generate calls have the same key, this leads to a potentially detectable situation if and only if the counter values used in the two Generate calls overlap. For AES, this occurs with a probability of  $2^{-116}$  for a 128-bit key; for three-key TDEA, the probability is  $2^{-57}$ . Thus, a total probability of detectable key collisions for the maximum size and number of Generate requests per DRBG instantiation using three-key TDEA is  $2^{-171}$ , and for AES is  $2^{-149}$  with 128-bit keys.

#### G.4 Dual\_EC\_DRBG

The **Dual\_EC\_DRBG** is the only DRBG mechanism in this Standard whose security is based on the difficulty of solving a known "hard" problem. That is, determining the internal state of the **Dual\_EC\_DRBG** from observed output is equivalent to solving the Elliptic Curve discrete log problem, for which no subexponential algorithms have been found, despite decades of effort.

Assuring that the internal state of a DRBG cannot be determined by observing some of its output is clearly the most important test that a deterministic algorithm must pass. Failure to do this would completely compromise the security of the cryptographic protocols that rely on the unpredictability of the unseen outputs. It is this feature that sets the **Dual\_EC\_DRBG** apart from the other DRBGs: no other DRBG mechanism in the Standard can relate the difficulty of determining its internal state from known outputs to a mathematically difficult problem.

However, there are other notions of "secure random" in the literature. In 1984, Blum and Micali introduced the notion of a *cryptographically secure random number generator*, defining it as one that passes the *next-bit test*: There exists no polynomial-time algorithm that, given  $n$  bits of output from the generator, can predict the  $(n+1)^{\text{st}}$  bit with a probability that is **significantly greater**<sup>5</sup> than  $1/2$ . The only DRBG mechanism that attempts to quantify its score on the *next-bit test* is the **Dual\_EC\_DRBG**.

The **Dual\_EC\_DRBG** produces bits in blocks: an (unknowable, initially non-deterministic)  $x$ -coordinate 's' on an elliptic curve over  $\text{GF}(p)$  is used as a scalar multiplier to jump to another point  $sP$  on the curve. Its  $x$ -coordinate, in turn, jumps to another point  $(sP)_x * Q$ . That point's  $x$ -coordinate is truncated by removing the high-order ' $d$ ' bits, and the remaining bits are output as a block of random bits. The process then iterates using  $s = (sP)_x$ , effecting a random walk on the curve. The value ' $d$ ' defaults to 16 (17 for the P251 curve, to get a multiple of 8), but can be varied by the implementation, as appropriate. The default was chosen as a compromise between the maximum efficiency ( $d=0$ ) and the maximum entropy ( $d=\text{curvesize}-1$ ).

The next-bit issue is addressed in Annex C. That discussion focuses on what is (by far) the worst case of the *next-bit test*: After all but one bit in a truncated  $x$ -coordinate is observed, the next bit can be predicted. A formula for the entropy in a truncated  $x$ -coordinate is derived, from which the probability of predicting that 'next bit' can be computed.

[Note that if the 'next bit' in a *next-bit test* occurs in the **next block** of output, the bit is part of a different  $x$ -coordinate on the curve. That there is no information about this bit from the previous  $x$ -coordinate can be inferred from the difficulty of the EC discrete log problem. So the only concern is about a 'next bit' in the same  $x$ -coordinate.]

For the P256 curve, Annex C provides a table of computed values using the entropy formula for ' $d$ ' between 0 and 16; the other P curves have essentially identical tables. The calculated entropy in each 240-bit block of the **Dual\_EC\_DRBG** output using the P256 curve is 239.9999890. This could be interpreted to say that, given 239 bits of a block, there are .0000110 bits of information about the 240-th bit. Said differently, more than 90,000 full blocks, or nearly 22 million bits, would have to be observed in order to distinguish **Dual\_EC\_DRBG** output from a true random source. If that's not sufficient for an application, the definition of the **Dual\_EC\_DRBG** allows ' $d$ ' to be increased, and the formula can be used to set the truncation parameter to whatever level of indistinguishability might be deemed to be needed. No other DRBG in this Standard addresses this issue.

Before resetting ' $d$ ' one might ask: "Should I be concerned about the .0000110 'missing' bits of entropy in blocks of the **Dual\_EC\_DRBG** output?" Firstly, most applications choosing to use this DRBG for key generation, key establishment and other cryptographic functions requiring secure random bits will not be hiding that fact. Thus, there will be no need to "distinguish" that the **Dual\_EC\_DRBG** is being used rather than some other DRBG. If the **Dual\_EC\_DRBG** output is being used as a one-time pad, that is, as a stream cipher, on a large amount of data, for which an adversary knows the data takes one of two values, then the adversary will be able to exploit the missing bits to determine from the ciphertext which of the two values the encrypted data is.

More importantly, 1 'missing' bit of entropy in 22,000,000 bits does not give a cryptanalyst any meaningful advantage in guessing a secret key comprised of such bits, certainly not a key of any reasonable size. Furthermore, there have not been any monobit or polybit biases found in **Dual\_EC\_DRBG** output. It is this type of bias that "Bleichenbacher"-type attacks use. [Granted, there is a tiny bias remaining in the truncated output blocks, due to the modular arithmetic. The

---

<sup>5</sup> The definition does not attempt to assign a numerical value to this term.

NIST primes used by this DRBG reduce the modular bias to a negligible size, and the truncation only reduces that further. This is addressed in Annex C.]

### ANNEX H: (Informative) Bibliography

- [1] Federal Information Processing Standard 186-3, *Digital Signature Standard (DSS)*, [Date to be inserted].
- [2] [Shparlinski] Mahassni, Edwin, and Shparlinski, Igor. On the Uniformity of Distribution of Congruential Generators over Elliptic Curves. Department of Computing, Macquarie University, NSW 2109, Australia; {eelmaha, igor}@isc.mq.edu.au.
- [3] Gurel, Nicholas, "Extracting Bits from Coordinates of a Point of an Elliptic Curve", Cryptology Eprint Archive: 2005/324.