DRAFT X9.82 (Random Number Generation)
Part 3, Deterministic Random Bit Generator Mechanisms

March 2006

Table of Contents

1 Scope 7

2 Conformance 7

3 Normative references 8

4 Definitions and Acronyms 8

5 Symbols 9

6 General Discussion and Organization 11

7 DRBG Functional Model 13

7.1 Functional Model 13

7.2 Functional Model Components 13

7.2.1 Entropy Input 13

7.2.2 Other Inputs 14

7.2.3 The Internal State 14

7.2.4 The DRBG Functions 14

8. DRBG Concepts and General Requirements 15

8.1 Introduction 15

8.2 DRBG Functions and a DRBG Instantiation 15

8.2.1 Functions 15

8.2.2 DRBG Instantiations 15

8.2.3 Internal States 15

8.2.4 Security Strengths Supported by an Instantiation 16

8.3 DRBG Boundaries 17

8.4 Seeds 19

8.4.1 General Discussion 19

8.4.2 Generation and Handling of Seeds 19

8.5 Other Inputs to the DRBG 22

8.5.1 Discussion 22

8.5.2 Personalization String 22

8.5.3 Additional Input 23

8.6 Prediction Resistance and Backtracking Resistance 23

9 DRBG Functions 24
9.1 General Discussion 24
9.2 Instantiating a DRBG 24
9.3 Reseeding a DRBG Instantiation 27
9.4 Generating Pseudorandom Bits Using a DRBG 29
9.4.1 The Generate Function 29
9.4.2 Reseeding at the End of the Seedlife 31
9.4.3 Handling Prediction Resistance Requests 32
9.5 Removing a DRBG Instantiation 32
9.6 Self-Testing of the DRBG (Health Testing) 33
9.6.1 Discussion 33
9.6.2 Testing the Instantiate Function 33
9.6.3 Testing the Generate Function 34
9.6.4 Testing the Reseed Function 34
9.6.5 Testing the Uninstantiate Function 35
9.7 Error Handling 35
9.7.1 General Discussion 35
9.7.2 Errors Encountered During Normal Operation 35
9.7.3 Errors Encountered During Self-Testing 35
10 DRBG Algorithm Specifications 36
10.1 Overview 36
10.2 Deterministic RBG Based on Hash Functions 36
10.2.1 Discussion 36
10.2.2 HMAC_DRBG (...) 37
10.2.2.1 Discussion 37
10.2.2.2 Specifications 38
10.3 DRBG Based on Block Ciphers 43
10.3.1 Discussion 43
10.3.2 CTR_DRBG 43
10.3.2.1 CTR_DRBG Description 43
10.3.2.2 Specifications 45

10.4 Deterministic RBG Based on Number Theoretic Problems 54
10.4.1 Discussion 54
10.4.2 Dual Elliptic Curve Deterministic RBG (Dual_EC_DRBG) 54
10.4.2.1 Discussion 54
10.4.2.2 Specifications 56
10.5 Auxilliary Functions 61
10.5.1 Discussion 61
10.5.2 Derivation Function Using a Hash Function (Hash_df) 61
10.5.3 Derivation Function Using a Block Cipher Algorithm (Block_Cipher_df) 62
10.5.4 Block_Cipher_Hash Function 64
11 Assurance 66
11.1 Overview 66
11.2 Minimal Documentation Requirements 67
11.3 Implementation Validation Testing 67
11.4 Health Testing 67
11.4.1 Overview 67
11.4.2 Known-Answer Testing 68
Annex A: (Normative) Application-Specific Constants 69
A.1 Constants for the Dual_EC_DRBG 69
A.1.1 Curves over Prime Fields 69
A.1.1.1 Curve P-256 69
A.1.1.2 Curve P-384 70
A.1.1.3 Curve P-521 70
A.2 Using Alternative Points in the Dual_EC_DRBG() 71
A.2.1 Generating Alternative *P,Q* 71
A.2.2 Additional Self-testing Required for Alternative *P,Q* 72
ANNEX B : (Normative) Conversion and Auxilliary Routines 73
B.1 Bitstring to an Integer 73
B.2 Integer to a Bitstring 73
B.3 Integer to an Octet String 73
B.4 Octet String to an Integer 74

Annex C: (Informative) Security Considerations 75
C.1 Extracting Bits in the Dual_EC_DRBG (...) 75
C.1.1 Potential Bias Due to Modular Arithmetic for Curves Over F_p 75
C.1.2 Adjusting for the Missing Bit(s) of Entropy in the x Coordinates 75
C.2 Reserve for a discussion of the nonce specified in Section 8.4.2, item 7 77
ANNEX D: (Informative) DRBG Selection 78
D.1 Choosing a DRBG Algorithm 78
D.2 HMAC_DRBG 78
D.3 CTR_DRBG 79
D.4 DRBGs Based on Hard Problems 80
ANNEX E: (Informative) Example Pseudocode for Each DRBG 82
E.1 Preliminaries 82
E.2 HMAC_DRBG Example 82
E.2.1 Discussion 82
E.2.2 Instantiation of HMAC_DRBG 83
E.2.3 Generating Pseudorandom Bits Using HMAC_DRBG 84
E.3 CTR_DRBG Example Using a Derivation Function 86
E.3.1 Discussion 86
E.3.2 The Update Function 86
E.3.3 Instantiation of CTR_DRBG Using a Derivation Function 87
E.3.4 Reseeding a CTR_DRBG Instantiation Using a Derivation Function 89
E.3.5 Generating Pseudorandom Bits Using CTR_DRBG 90
E.4 CTR_DRBG Example Without a Derivation Function 92
E.4.1 Discussion 92
E.4.2 The Update Function 92
E.4.3 Instantiation of CTR_DRBG Without a Derivation Function 92
E.4.4 Reseeding a CTR_DRBG Instantiation Without a Derivation Function 93
E.4.5 Generating Pseudorandom Bits Using CTR_DRBG 93
E.5 Dual_EC_DRBG Example 93
E.5.1 Discussion 93
E.5.2 Instantiation of Dual_EC_DRBG 94

E.5.3 Reseeding a Dual_EC_DRBG Instantiation 96
E.5.4 Generating Pseudorandom Bits Using Dual_EC_DRBG 97
ANNEX F: (Informative) DRBG Provision of RBG Security Properties 99
F.1 Introduction 99
F.2 Security Strengths 99
F.3 Entropy and Min-Entropy 99
F.4 Backtracking Resistance and Prediction Resistance 99
F.5 Indistinguishability and Unpredictability 99
F.6 Desired RBG Output Properties 100
F.7 Desired RBG Operational Properties 100
ANNEX G: (Informative) Bibliography 102

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 defines techniques for the generation of random bits using deterministic methods. This part includes:

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

This part of ANS X9.82 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 their 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 deterministic random bit generator (DRBG) may claim conformance with ANS X9.82 if it implements 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.

Conformance can 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 X9.62-2006, *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 Definitions and Acronyms

Definitions used in this part of ANS X9.82 are provided in Part 1.

The following abbreviations are used in this document:

Abbreviation	Meaning
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.

5 Symbols

The following symbols are used in this document.

Symbol	Meaning
+	Addition
$\lceil X \rceil$	Ceiling: the smallest integer $r \geq X$. 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 mod 2) of two bitstrings X and Y of the same length.
$X \mid\mid Y$	Concatenation of two strings X and Y. X and Y are either both bitstrings, or both octet strings.
gcd (*x*, y)	The greatest common divisor of the integers x and y.
len (*a*)	The length in bits of string a.
x mod n	The unique remainder r (where $0 \leq r \leq n-1$) when integer x is divided by n. For example, 23 mod 7 = 2.
	Used in a figure to illustrate a "switch" between sources of input.

$\{a_1, \ldots a_i\}$	The internal state of the DRBG at a point in time. The types and number of the a_i depends on the specific DRBG.
0x*ab*	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,…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 level 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), an implementation cannot achieve the properties specified in Part 1 unless the entropy input source 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 entropy input source must be considered.

Throughout the remainder of this document, the term "DRBG mechanism" has been shortened to "DRBG".

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

— Section 7 provides a functional model for a DRBG that particularizes the general functional model of Part 1.

— Section 8 provides DRBG concepts and general requirements.

— Section 9 specifies the DRBG functions that will be used to access the DRBG algorithms specified in Section 10.

— Section 10 specifies Approved DRBG algorithms.

— Section 11 addresses assurance issues for DRBGs.

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 DRBGs.

— Annex D provides a discussion on DRBG selection.

— Annex E provides example pseudocode for each DRBG.

— 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 DRBG Functional Model

7.1 Functional Model

Part 1 of this Standard provides a general functional model for random bit generators (RBGs). Figure 1 particularizes the functional model of Part 1 for DRBGs. The components of this model are discussed in the following subsections.

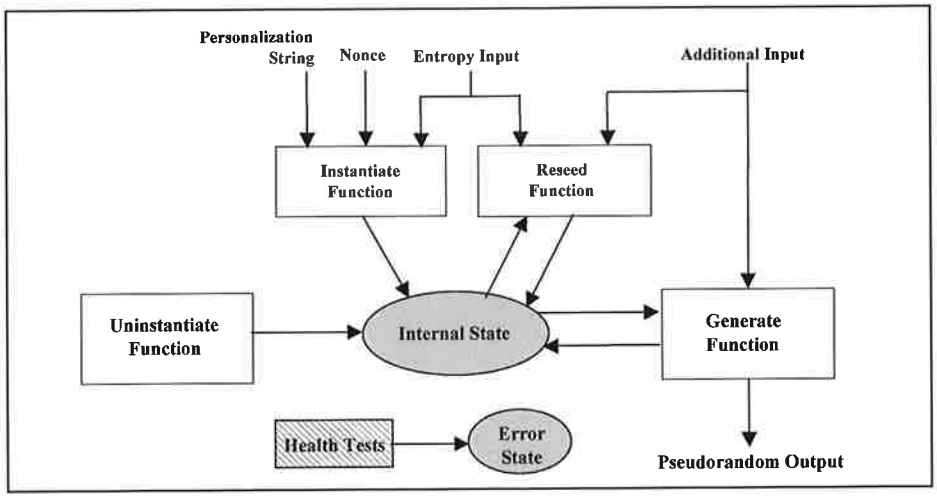

Figure 1: DRBG Functional Model

7.2 Functional Model Components

7.2.1 Entropy Input

The entropy input is provided to a DRBG 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 requested amount of entropy for a DRBG. Appropriate sources for the entropy input are discussed in Parts 2 and 4 of this Standard.

Ideally, the entropy input will be full entropy; however, the DRBGs 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 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 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.

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 bits and a nonce to create the initial DRBG seed. The personalization string **shall** be unique for all instantiations of the same DRBG 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 uses or acts upon. The internal state contains both administrative data (e.g., the security level) and data that is acted upon and/or modified during the generation of pseudorandom bits (i.e., the *working state*). The contents of the internal state is dependent on the specific DRBG and includes all information that is required to produce the pseudorandom bits from one request to the next.

7.2.4 The DRBG Functions

The DRBG functions handle the DRBG's internal state. The DRBGs 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 uninstantiate function zeroizes (i.e., erases) the internal state.
5. The health test function determines that the DRBG continues to function correctly.

8. DRBG Concepts and General Requirements

8.1 Introduction

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

8.2 DRBG Functions and a DRBG Instantiation

8.2.1 Functions

A DRBG requires instantiate, uninstantiate, generate, and health testing functions. A DRBG 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).

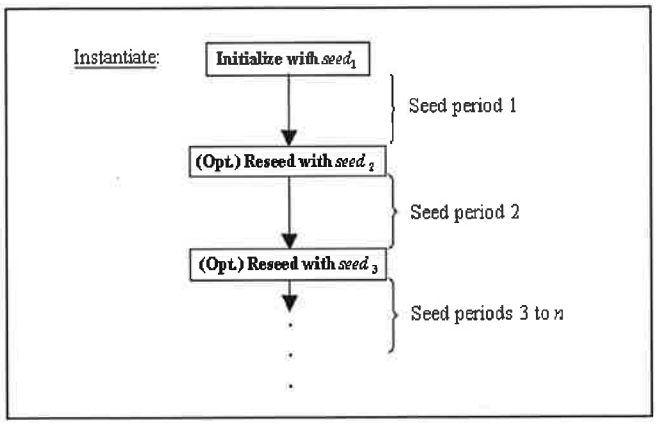

Figure 2: DRBG Instantiation

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 must usually 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).

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 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 DRBGs specified in this Standard support four security strengths: 112, 128, 192 or 256 bits. 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.

Table 1: Possible Instantiated Security Strengths

Maximum Designed Security Strength	**112**	**128**	**192**	**256**
Possible Instantiated Security Strengths	112	112, 128	112, 128, 192	112, 128, 192, 256

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, 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 96 bits will result in an instantiation at the 112-bit security strength).

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 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 Boundaries

As a convenience, this Standard uses the notion of a "DRBG boundary" to explain the operations of a DRBG and its interaction with and relation to other processes; a DRBG boundary contains all DRBG functions and internal states required for a DRBG. A DRBG boundary is entered via the DRBG's public interfaces, which are made available to consuming applications.

Within a DRBG boundary,

1. The DRBG internal state and the operation of the DRBG functions **shall** only be affected according to the DRBG specification.
2. The DRBG internal state **shall** exist solely within the DRBG boundary. The internal state **shall** be contained within the DRBG boundary and **shall not** be accessed by non-DRBG functions.
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 boundary, except as specified for the DRBG pseudorandom bit outputs.

Each DRBG 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 functions are not affected.

A DRBG'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 functions that are distributed across multiple devices. In this latter case, each device has a DRBG sub-boundary that contains the DRBG functions implemented on that device, and the boundary around the entire DRBG consists of the aggregation of sub-boundaries providing the DRBG functionality. The use of distributed DRBG 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.

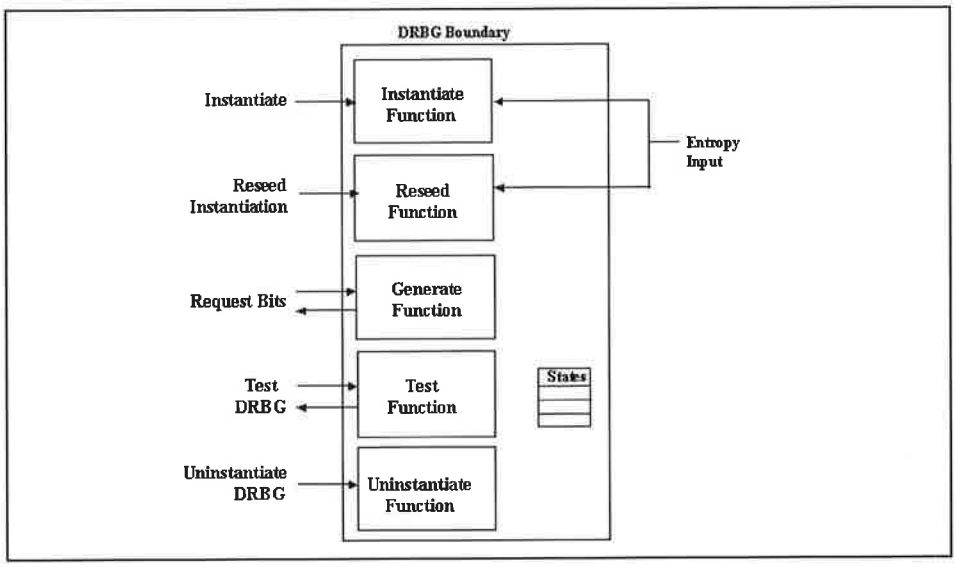

Figure 3: DRBG Functions Within a Single Device

Although the entropy input that is used to create the seed is shown in the figures as originating outside the DRBG boundary, it may originate from within the boundary.

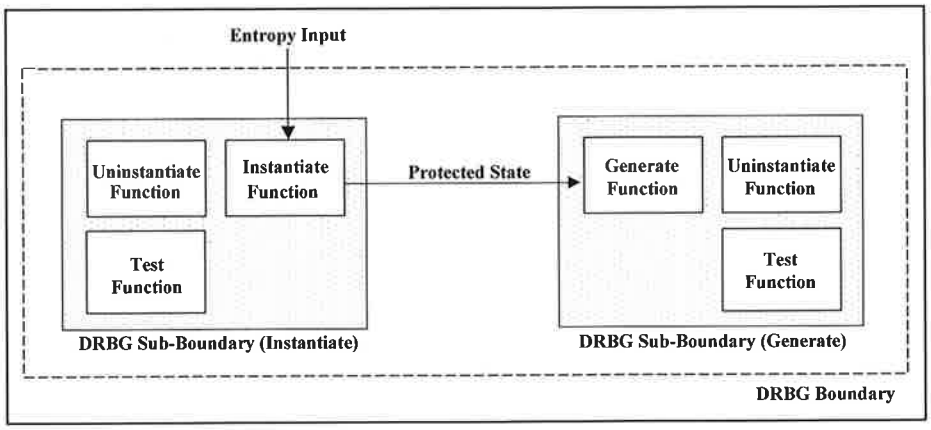

Figure 4: Distributed DRBG Functions

Each DRBG boundary or sub-boundary **shall** contain a test function to test the "health" of other DRBG 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 functions are distributed, appropriate mechanisms **shall** be used to protect the confidentiality and integrity of the internal state or parts of the internal state that are transferred between the distributed DRBG sub-boundaries. The confidentiality and integrity mechanisms and security strength **shall** be consistent with the data to be protected by the DRBG's consuming application (see ASC X9 Registry).

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 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. This Standard also recommends the inclusion of a personalization string; requirements for the personalization string are discussed in Section 8.5.2.

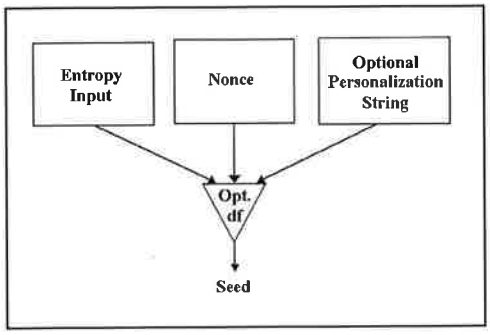

Figure 5: Seed Construction for Instantiation

Depending on the DRBG 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 based on block cipher algorithms (see Section 10.3) may be implemented without a derivation function. When implemented in this manner, a nonce (as shown in Figure 5) is not used. Note, however, that the personalization string could contain a nonce, if desired.

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[1], 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.

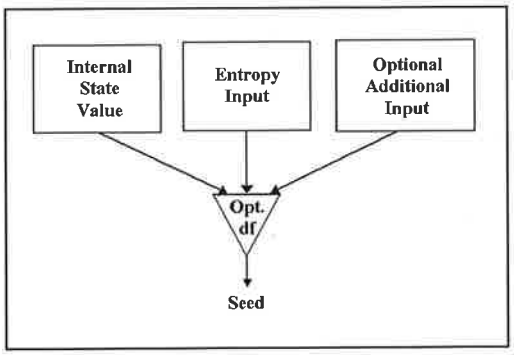

Figure 6: Seed Construction for Reseeding

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 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 instantiatiation, 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 and the security strength required by the consuming application. See Section 10.

5. Entropy input source: The source of the entropy input **shall** be either:

    a. An Approved NRBG,

[1] See each DRBG specification for the value that is used.

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 entropy input source 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 instantation in order to provide a security cushion to block certain attacks. The nonce **shall** be either:

   a. A random value with at least (*security_strength*/2) bits of entropy,

   b. A non-random value that is expected to repeat no more often than a (*security_strength*/2)-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 *security_strength*) bits.

   The nonce is required for instantiation to provide *security_strength* bits of security. 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). Further discussion is provided in Annex C.2.

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 length of the seed period); the maximum seedlife is dependent on the DRBG 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. 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 there is an undetected failure in the entropy input source, a reseeded DRBG instantiation will still retain any previous entropy, whereas a newly instantiated DRBG may not have sufficient entropy to support the requested security strength.

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

   A DRBG does not provide output until a seed is available, and the internal state has been initialized.

10. Seed separation: Seeds used by DRBGs and the entropy input used to create those seeds **shall not** be used for other purposes (e.g., domain parameter or prime number generation).

8.5 Other Inputs to the DRBG

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 DRBG or an NRBG.

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. The value of any secret information contained in the personalization string **should** be no greater than the claimed strength of the DRBG, as the DRBG's cryptographic mechanisms (specifically, its backtracking resistance and the entropy provided in the entropy input) will protect this information from disclosure. Good choices for the personalization string contents include:

- Device serial numbers,
- Public keys,

- User identification,
- Private keys,
- PINs and passwords,
- 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, and
- 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. 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. Prediction resistance can be provided only by ensuring that a DRBG is effectively reseeded between DRBG requests. The DRBGs 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 Functions

9.1 General Discussion

The DRBG 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.

In the specifications of this Standard, a **Get_entropy_input** pseudo-function is used for convenience. This 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*) = **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 DRBGs, the *min_length* = *min_entropy* for the **Get_entropy_input** function, in which case, the second parameter could be omitted.

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 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. Returns a *state_handle* for the internal state to the consuming application (see below).

Let *working_state* be the working state for the particular DRBG, and let *min_length*, *max_length*, and *highest_supported_security_strength* be defined for each DRBG (see Section 10). The following or an equivalent process **shall** be used to instantiate a DRBG.

Input from a consuming application for instantiation:

1. *requested_instantiation_security_strength*: A requested security strength for the instantiation. DRBG implementations that support only one security strength do not require this parameter; however, any application using that DRBG implementation must be aware of this limitation.
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. DRBGs that are implemented to always 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 a DRBG implementation. If the *prediction_resistance_flag* is not needed (i.e., because prediction resistance is always or never performed), then the input parameter may be omitted, and the *prediction_resistance_flag* may be 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 (see Section 10). If a personalization string will never be used, then the input parameter and step 3 of the instantiate process may be omitted, and instantiate process step 9 may be modified to omit the personalization string.

Required information not provided by the consuming application during instantiation:

Comment: 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 (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 would be adjusted to include the entropy for the *nonce* (i.e., *security_strength* would be increased by at least *security_strength*/2), process step 8 would be omitted, and the *nonce* would be omitted from the parameter list in 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.

Information retained within the DRBG boundary after instantiation:

The internal state for the DRBG, including the *working_state* and administrative information (see Sections 8.2.3 and 10).

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 **should** be omitted.

5. Using the *security_strength*, select appropriate DRBG 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 reseeding 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 available, 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, and let *min_length* and *max_ length* be defined for each DRBG (see Section 10).

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

Input from a consuming application for reseeding:

1) *state_handle*: A pointer or index that indicates the internal state to be reseeded. This value was returned from the instantiate function specified in Section 9.2.
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 (see Section 10). If *additional_input* will never be used, then the input parameter and step 2 of the reseed process may be omitted, and step 5 may be modified to remove the *additional_input* from the parameter list.

Required information not provided by the consuming application during reseeding:

Comment: This input **shall not** be provided by the consuming application in the input parameters.

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 (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 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).

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

Input from a consuming application for generation:

1. *state_handle*: A pointer or index that indicates the internal state to be used.
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.
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 this limitation.
4. *prediction_resistance_request*: Indicates whether or not prediction resistance is to be provided. DRBGs that are implemented to always 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 application before electing to use such a DRBG implementation.

   If prediction resistance is never provided, then the *prediction_resistance_request* input parameter and step 5 of the generate process may be omitted, and step 7 may be modified to omit the check for the *prediction_resistance_request*.

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

status = **Reseed** (*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 *additional_input* is never provided, 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 (see Section 10). If *additional_input* will never be used, then the input parameter, process step 4, step 7.4 and the *additional_input* input parameter in steps 7.1 and 8 may be 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 function. The *status* will indicate **SUCCESS** or an **ERROR**.
2. *pseudorandom_bits*: The pseudorandom bits that were requested.

Information retained within the DRBG 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** (*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 available, then steps 6 and 7 may be 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** (*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 function (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, it is clear, so step 7 would be 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 will check 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. 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 step 7, resulting in a reseed, a new internal state for the instantiation and a *Null* value for any additional input provided during the generate request; obtains pseudorandom bits (see step 8); passes through step 9, since another reseed will not be required; and continues with 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., uninstantiate) a DRBG instantiation:

Input from a consuming application for uninstantiation:

1. *state_handle*: A pointer or index that indicates the internal state to be "released".

Output to a consuming application after uninstantiation:

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

Information retained within the DRBG boundary after uninstantiation:

An empty internal state.

Uninstantiate 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**.

9.6 Self-Testing of the DRBG (Health Testing)

9.6.1 Discussion

A DRBG **shall** perform self testing to obtain assurance that the implementation continues to operate as designed and implemented (health testing). The testing function(s) within a DRBG boundary (or sub-boundary) **shall** test each DRBG function within that boundary. Note that this may require the creation and use of an instantiation for testing purposes only.

Errors occurring during testing **shall** be perceived as catastrophic DRBG failures (see Section 9.7.3). The condition causing the failure **shall** be corrected and the DRBG re-instantiated before requesting pseudorandom bits (also, see Section 9.7)

9.6.2 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 input 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 allowed) **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 entropy input source 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.

9.6.3 Testing the Generate Function

Known-answer tests **shall** be performed on the generate function before the first use of the function 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 allowed) and the working state of the internal state value (see Sections 8.2.3 and 10) **shall** be used. If prediction resistance is available, 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.

9.6.4 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 allowed) 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 available in an implementation, the reseed function **shall** be tested whenever the generate function is tested (see above).
2. When prediction resistance is not available 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.

9.6.5 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 zeroized.

9.7 Error Handling

9.7.1 General Discussion

The expected errors are indicated for each DRBG 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.

9.7.2 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.

For catastrophic errors (i.e., those errors indicated by the **CATASTROPHIC_ERROR_FLAG** in the pseudocode), the DRBG **shall not** produce further output until the source of the error is corrected, and the DRBG is re-instantiated.

9.7.3 Errors Encountered During Self-Testing

During self-testing, all unexpected behavior is catastrophic. The DRBG **shall** be corrected, and the DRBG **shall** be re-instantiated before the DRBG can be used to produce pseudorandom bits. Examples of unexpected behavior include:

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

10 DRBG Algorithm Specifications

10.1 Overview

Several DRBGs are specified in this Standard. The selection of a DRBG 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. A detailed discussion on DRBG selection is provided in Annex D. Pseudocode examples for each DRBG are provided in Annex E. Conversion specifications required for the DRBG 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 hash DRBG is based on a hash function that is non-invertible or one-way. The hash-based DRBG 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 hash function is provided in ASC X9 Registry 0003. 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

	SHA-1	**SHA-224**	**SHA-256**	**SHA-384**	**SHA-512**
Supported security strengths	See ASC X9 Registry 0003				
highest_supported_security_strength	See ASC X9 Registry 0003				
Output Block Length (*outlen*)	**160**	224	256	384	512
Required minimum entropy for instantiate and reseed	*security_strength*				
Minimum entropy input length (*min_length*)	*security_strength*				
Maximum entropy input length (*max_ length*)	$\leq 2^{35}$ bits				
Seed length (*seedlen*)	**440**	440	440	888	888
Maximum personalization string length (*max_personalization_string_length*)	$\leq 2^{35}$ bits				

	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512
Maximum additional_input length (*max_additional_input_length*)	$\leq 2^{35}$ bits				
max_number_of_bits_per_request	$\leq 2^{19}$ bits				
Number of requests between reseeds (*reseed_interval*)	$\leq 2^{48}$				

Note that since SHA-224 is based on SHA-256, there is no efficiency benefit for using the SHA-224; this is also the case for SHA-384 and SHA-512, i.e., the use of SHA-256 or SHA-512 instead of SHA-224 or SHA-384, respectively, is preferred. 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 uses the **Update** function specified in Section 10.2.2.2 and the **HMAC** function within the **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 (**Update**). This function is called by the **HMAC_DRBG** instantiate, generate and reseed algorithms to adjust the internal state when new entropy or additional

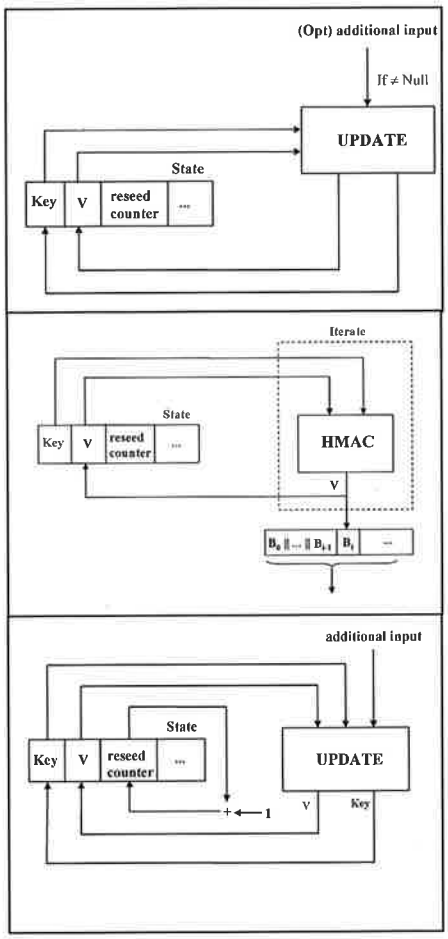

Figure 7: HMAC_DRBG Generate Function

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 **Update** 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 generates pseudorandom bits.
   c. A counter (*reseed_counter*) that indicates the number of requests for pseudorandom bits since instantiation or reseeding.

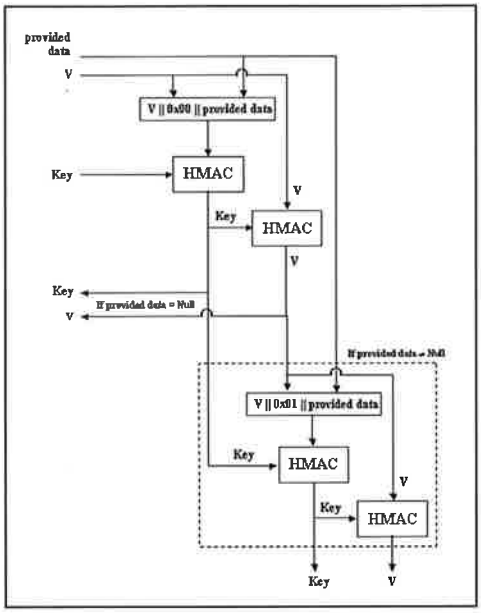

Figure 8: HMAC_DRBG Update Function

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.

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

10.2.2.2.2 The Update Function (Update)

The **Update** function updates the internal state of **HMAC_DRBG** using the *provided_data*. Note that for this DRBG, the **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 from Table 2 in Section 10.2.1.

The following or an equivalent process **shall** be used as the **Update** function.

Input:

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* = **HMAC** (*K*, *V* || 0x00 || *provided_data*).
2. *V* = **HMAC** (*K*, *V*).
3. If (*provided_data* = *Null*), then return *K* and *V*.
4. *K* = **HMAC** (*K*, *V* || 0x01 || *provided_data*).
5. *V* = **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, step 5 of the instantiate process **should** be 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 **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 (see step 9 of the instantiate process in Section 9.2):

Input:

1. *entropy_input*: The string of bits obtained from the entropy input source.
2. *nonce*: A string of bits as specified in Section 8.4.2.
3. *personalization_string*: The personalization string received from the consuming application. If a *personalization_string* will never be used, then step 1 may be modified to remove the *personalization_string*.

Output:

1. *initial_working_state*: The inital values for *V*, *Key* and *reseed_counter* (see Section 10.2.2.2.1).

HMAC_DRBG Instantiate Process:

1. *seed_material* = *entropy_input* || *nonce* || *personalization_string*.
2. *Key* = 0x00 00...00. Comment: *outlen* bits.
3. *V* = 0x01 01...01. Comment: *outlen* bits.

   Comment: Update *Key* and *V*.
4. (*Key*, *V*) = **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 **Update** be the function specified in Section 10.2.2.2.2. The following process or its equivalent **shall** be used as the reseed algorithmn for this DRBG (see step 5 of the reseed process in Section 9.3):

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 entropy input source.
3. *additional_input*: The additional input string received from the consuming application. If *additional_input* will never be used, then process step 1 may be 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* || *additional_input*.
2. (*Key*, *V*) = **Update** (*seed_material*, *Key_old*, *V_old*).
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 *max_number_of_bits_per_request* and *outlen* 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. 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 (see step 8 of the generate process in Section 9.4):

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 an implementation will never use *additional_input*, then step 3 of the HMAC generate process may be omitted. If an implementation does not include the *additional_input* parameter as one of the calling parameters, or if the implementation allows *additional_input*, but a given request does not provide any *additional_input*, then a *Null* string **shall** be used as the *additional_input* in step 6.

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*) = **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) = **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 Based on Block Ciphers

10.3.1 Discussion

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

10.3.2 CTR_DRBG

10.3.2.1 CTR_DRBG Description

CTR_DRBG uses an Approved block cipher algorithm in the counter mode as specified in 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 (**Update**). Figure 9 depicts the **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.

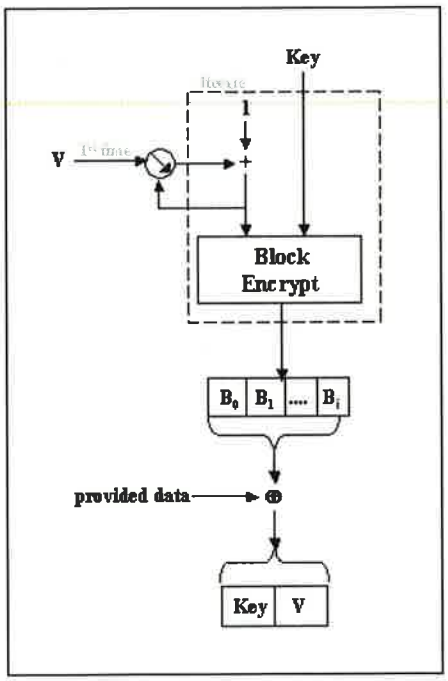

Figure 9: CTR_DRBG Update Function

Comment [ebb1]: This only applies to AES, not TDEA. How should this be handled ?

Table 3: Definitions for the CTR_DRBG

	3 Key TDEA	AES-128	AES-192	AES-256
Supported security strengths	See ASC X9 Registry			
highest_supported_security_strength	See ASC X9 Registry			
Output block length (*outlen*)	64	128	128	128

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

The **CTR_ DRBG** may be implemented to use the block cipher derivation function specified in Section 10.5.2 during instantiation and reseeding. However, the DRBG 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 functon **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[2] (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).

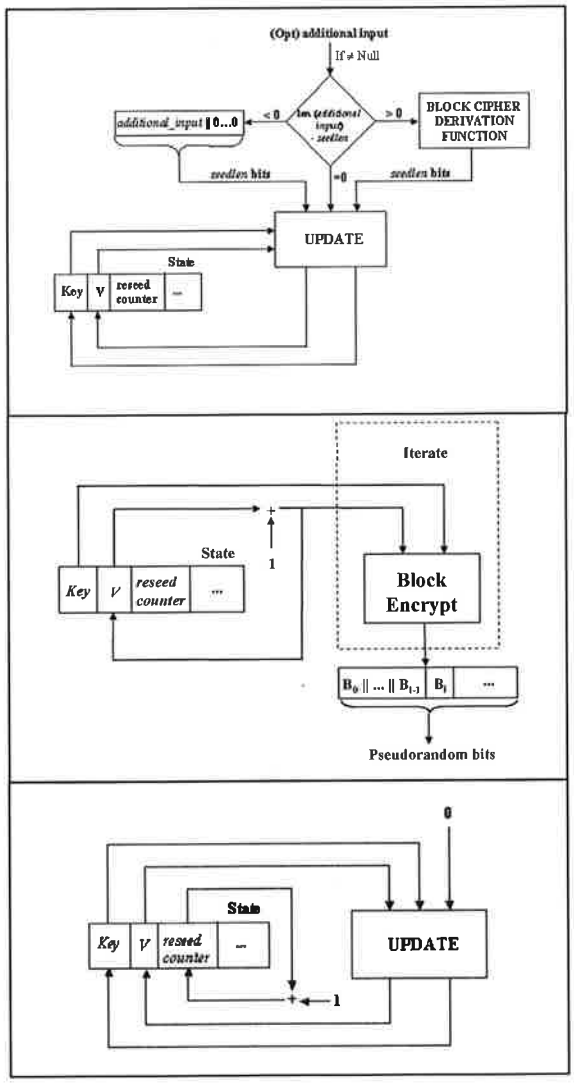

Figure 10: CTR-DRBG

[2] The specifications in this Standard do not accommodate the special treatment required for a nonce in this case.

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:

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.

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

10.3.2.2.2 The Update Function (Update)

The **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 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 **Update** function:

Input:

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* = **Block_Encrypt** (*Key*, *V*).

   2.3 *temp* = *temp* || *ouput_block.*
3. *temp* = Leftmost *seedlen* bits of *temp.*

4 *temp* = *temp* ⊕ *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, step 5 of the instantiate function **should** be 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:

Let **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.

For this DRBG, there are two cases for the processing. The input to the instantiate algorithm is the same for each case; likewise for the output from the instantiate algorithm. However, the process steps are slightly different (see Sections 10.3.2.2.3.1 and 10.3.2.2.3.2).

Input:

1. *entropy_input*: The string of bits obtained from the entropy input source.
2. *nonce*: A string of bits as specified in Section 8.4.2; this string **shall not** be present unless a derivation function is used.
3. *personalization_string*: The personalization string received from the consuming application.

Output:

1. *initial_working_state*: The inital values for *V*, *Key*, and *reseed_counter* (see Section 10.3.2.2.1).

10.3.2.2.3.1 The Process Steps for Instantiation When a Derivation Function is Not Used

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

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* ⊕ *personalization_string*.
4. *Key* = 0^{keylen}. Comment: *keylen* bits of zeros.
5. *V* = 0^{outlen}. Comment: *outlen* bits of zeros.
6. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).
7. *reseed_counter* = 1.
8. **Return** *V*, *Key*, and *reseed_counter* as the *initial_working_state*.

Implementation note:

If a *personalization_string* will never be provided from the instantiate function, then steps 1-3 are replaced by:

seed_material = *entropy_input*.

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

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:

CTR_DRBG Instantiate Process:

1. *seed_material* = *entropy_input* || *nonce* || *personalization_string*.

   Comment: Ensure that the length of the *seed_material* is exactly *seedlen* bits.
2. *seed_material* = **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*) = **Update** (*seed_material*, *Key*, *V*).
6. *reseed_counter* = 1.
7. **Return** *V*, *Key*, and *reseed_counter* as the *initial_working_state*.

Implementation note:

If a *personalization_string* will never be provided from the instantiate function, then steps 1-2 are replaced by:

seed_material = **Block_Cipher_df** (*entropy_input*, *seedlen*).

10.3.2.2.4 Reseeding a CTR_DRBG Instantiation

Notes for the reseed function specified in Section 9.3:

The reseeding 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:

Let **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.

For this DRBG, there are two cases for the processing. The input to the reseed algorithm is the same for each case; likewise for the output from the reseed algorithm. However, the process steps are slightly different (see Sections 10.3.2.2.4.1 and 10.3.2.2.4.2).

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 entropy input source.
3. *additional_input*: The additional input string received from the consuming application.

Output :

1. *new_working_state*: The new values for *V*, *Key*, and *reseed_counter*.

10.3.2.2.4.1 The Process Steps for Reseeding When a Derivation Function is Not Used

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

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* || $0^{seedlen - temp}$.
3. *seed_material* = *entropy_input* ⊕ *additional_input*.
4. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).

5. *reseed_counter* = 1.
6. **Return** *V*, *Key* and *reseed_counter* as the *new_working_state*.

Implementation note:

If *additional_input* will never be provided from the reseed function, then steps 1-3 are replaced by:

seed_material = *entropy_input*.

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

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 (see step 5 of Section 9.3):

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*) = **Update** (*seed_material*, *Key*, *V*).
4. *reseed_counter* = 1.
5. **Return** *V*, *Key*, and *reseed_counter* as the *new_working_state*.

Implementation note:

If *additional_input* will never be provided from the reseed function, then steps 1-2 become:

seed_material = **Block_Cipher_df** (*entropy_input*, *seedlen*).

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, there are two cases for the processing. The input to the generate algorithm is the same for each case; likewise for the output from the generate

algorithm. However, the process steps are slightly different (see Sections 10.3.2.2.5.1 and 10.3.2.2.5.2).

Let **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.

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 3 becomes:

$$additional_input = 0^{seedlen}.$$

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*.

10.3.2.2.5.1 The Process Steps for Generating Pseudorandom BitsWhen 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 (see step 8 of the generate process in Section 9.4.1):

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* || $0^{seedlen - temp}$.

2.3 (*Key*, *V*) = **Update** (*additional_input*, *Key*, *V*).

Else *additional_input* = $0^{seedlen}$.

3. *temp* = *Null*.
4. While (**len** (*temp*) < *requested_number_of_bits*) do:

   4.1 $V = (V + 1) \bmod 2^{outlen}$.

   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*) = **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*.

10.3.2.2.5.2 The Process Steps for Generating Pseudorandom BitsWhen 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 (see step 8 of the generate process in Section 9.4.1):

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

   2.1 *additional_input* = **Block_Cipher_df** (*additional_input*, *seedlen*).

   2.2 (*Key*, *V*) = **Update** (*additional_input*, *Key*, *V*).

   Else *additional_input* = $0^{seedlen}$.
3. *temp* = *Null*.
4. While (**len** (*temp*) < *requested_number_of_bits*) do:

   4.1 $V = (V + 1) \bmod 2^{outlen}$.

   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*) = **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 Deterministic RBG 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 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 a seed that is m bits in length (i.e., *seedlen* = m) 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 of the NIST curves given in this Standard for the DRBG, $m \geq 256$. Figure 11 depicts the **Dual_EC_DRBG**.

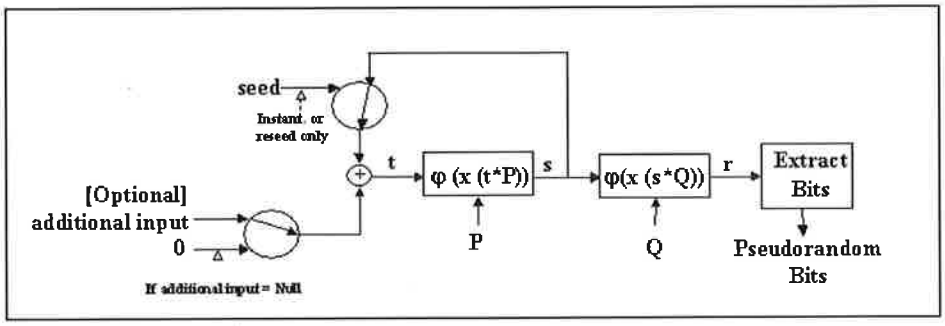

Figure 11: Dual_EC_DRBG

The instantiation of this DRBG 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 **shall** have entropy that is at least *security_strength* bits. Further requirements for the *seed* are provided in Section 8.2. This DRBG 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 $i = 1,2,3,\ldots$, as follows:

$S_i = \varphi(x(S_{i-1} * P))$

$R_i = \varphi(x(S_i * Q))$.

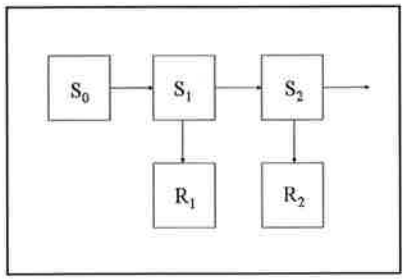

Figure 12: Dual_EC_DRBG (...) Backtracking Resistance

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 φ 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 resistence 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.

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 its 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**
Supported security strengths	See SP 800-57		
Size of the base field (in bits)	256	384	521
highest_supported_ security_strength	See SP 800-57		
Output block length (*max_outlen* = largest multiple of 8 less than (size of the base field) - (13 + $\log_2$ (the cofactor))	240	368	504
Required minimum entropy for instantiate and reseed	*security_strength*		
Minimum entropy input length (*min_length*)	*security_strength*		
Maximum entropy input length (*max_length*)	$\leq 2^{13}$ bits		

	P-256	P-384	P-521
Maximum personalization string length (*max_personalization_string_length*)	$\leq 2^{13}$ bits		
Maximum additional input length (*max_additional_input_length*)	$\leq 2^{13}$ bits		
Seed length (*seedlen*)	**2 × *security_strength***		
Appropriate hash functions	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512	SHA-224, SHA-256, SHA-384, SHA-512	SHA-256, SHA-384, SHA-512
max_number_of_bits_per_request	*max_outlen* × *reseed_interval*		
Number of blocks between reseeding (*reseed_interval*)	$\leq 2^{32}$ 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 (*seedlen*, p, a, b, n), where *seedlen* is the length of the seed, p is the order of the 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; 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 (*block_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,
   b. A *prediction_resistance_flag* that indicates whether prediction resistance is required by the DRBG.

The value of s is the critical value of the internal state upon which the security of this DRBG 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 ≥ *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 (see step 9 of the instantiate process in Section 9.2):

Input:

1. *entropy_input*: The string of bits obtained from the entropy input source.
2. *nonce*: A string of bits as specified in Section 8.4.2.
3. *personalization_string*: The personalization string received from the consuming application.

Output:

1. s: The initial secret value for the *initial_working_state*.
2. *block_counter*: The initialized block counter for reseeding.

Dual_EC_DRBG Instantiate Process:

1. *seed_material* = *entropy_input* || *nonce* || *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* = **Hash_df** (*seed_material*, *seedlen*).
3. *block_counter* = 0.
4. Return *s*, and *block_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):

Input:

1. *s*: The current value of the secret parameter in the *working_state*.
2. *entropy_input*: The string of bits obtained from the entropy input source.
3. *additional_input*: The additional input string received from the consuming application.

Output:

1. *s*: The new value of the secret parameter in the *new_working_state*.
2. *block_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* = **pad8** (*s*) || *entropy_input* || *additional_input*.
2. *s* = **Hash_df** (*seed_material*, *seedlen*).
3. *block_counter* = 0.
4. Return *s* and *block_counter* for the *new_working_state*.

Implementation notes:

If an implementation never allows *additional_input*, then step 1 may be modified as follows :

seed_material = **pad8** (*s*) || *entropy_input.*

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.

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. $\varphi(x)$ 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 $\varphi(x)$ 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: $c_{m-1} \| c_{m-2} \| \ldots \| c_1 \| c_0$, a bitstring, with c_{m-1} being leftmost

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

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

Thus, any field element x of the form Fa 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):

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.

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. *block_counter*: The updated block counter for reseeding.

Dual_EC_DRBG Generate Process:

Comment: Check whether a reseed is required.

1. If $\left(block_counter + \left\lceil \frac{requested_number_of_bits}{outlen} \right\rceil\right) > reseed_interval$, 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. $t = s \oplus additional_input$. 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. $s = \varphi(x(t * P))$. Comment: s is a *seedlen*-bit number.
7. $r = \varphi(x(s * Q))$. Comment: r is a *seedlen*-bit number.
8. *temp* = *temp* || (**rightmost** *outlen* bits of r).
9. *additional_input*=0 Comment: *seedlen* zeroes; *additional_input_string* is added only on the first iteration.
10. *block_counter* = *block_counter* + 1.
11. $i = i + 1$.
12. If (**len** (*temp*) < *requested_number_of_bits*), then go to step 5.
13. *returned_bits* = **Truncate** (*temp*, $i \times$ *outlen*, *requested_number_of_bits*).
14. Return **SUCCESS**, *returned_bits*, and s, and *block_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.

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, and let *outlen* be its output length.

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

Input:

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 × *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. If *no_of_bits_to_return* > *max_number_of_bits*, then return an **ERROR-_FLAG**.
2. *temp* = the Null string.
3. $len = \left\lceil \frac{no_of_bits_to_return}{outlen} \right\rceil$.
4. *counter* = an 8-bit binary value representing the integer "1".
5. For *i* = 1 to *len* do

   Comment : In step 5.1, *no_of_bits_to_return* is used as a 32-bit string.

   5.1 *temp* = *temp* || **Hash** (*counter* || *no_of_bits_to_return* || *input_string*).

   5.2 *counter* = *counter* + 1.
6. *requested_bits* = Leftmost (*no_of_bits_to_return*) of *temp*.
7. 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 **Block_Cipher_Hash** 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.

Input:

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

   Comment: L is the bitstring represention of the integer resulting from **len** (*input_string*)/8. L **shall** be represented as a 32-bit integer.
3. N = *number_of_bits_to_return*/8.

   Comment : N is the bitstring represention of the integer resulting from *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*.
3. $S = L \| N \| input_string \| 0x80$.

   Comment : Pad S with zeros, if necessary.
4. While (**len** (S) mod *outlen*) $\neq 0$, $S = S \| 0x00$.

   Comment : Compute the starting value.
5. *temp* = the *Null* string.
6. $i = 0$.

   Comment : i **shall** be represented as a 32-bit integer, i.e., **len** (i) = 32.
7. K = Leftmost *keylen* bits of 0x00010203...1F.
8. While **len** (*temp*) < *keylen* + *outlen*, do

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

   Comment: The 32-bit integer represenation of i is padded with zeros to *outlen* bits.

   8.2 *temp* = *temp* || **Block_Cipher_Hash** (K, ($IV \| S$)).

   8.3 $i = i + 1$.

   Comment: Compute the requested number of bits.
9. K = Leftmost *keylen* bits of *temp*.
10. X = Next *outlen* bits of *temp*.
11. *temp* = the *Null* string.

12. While **len** (*temp*) < *number_of_bits_to_return*, do

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

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

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

14. Return **SUCCESS** and *requested_bits*.

10.5.4 Block_Cipher_Hash Function

The **Block_Encrypt** pseudo-function is used for convenience in the specification of the **Block_Cipher_Hash** 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:

output_block = **Block_Encrypt** (*Key*, *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 **Block_Cipher_Hash** 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.

Input:

1. *Key*: The key to be used for the block cipher opeation.
2. *data_to_hash*: The data to be operated upon. Note that the length of *data_to_hash* must be a multiple of *outlen*. This is guaranteed by 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.

Block_Cipher_Hash process:

1. *chaining_value* = 0^{outlen}. Comment: Set the first chaining value to *outlen* zeros.
2. n = **len** (*data_to_hash*)/*outlen*.
3. Split the *data_to_hash* into n blocks of *outlen* bits each forming $block_1$ to $block_n$.
4. For i = 1 to n do

4.1 *input_block* = *chaining_value* $\oplus$ $block_i$.

4.2 *chaining_value* = **Block_Encrypt** (*Key*, *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 DRBGs in this Standard is depicted in Figure 13.

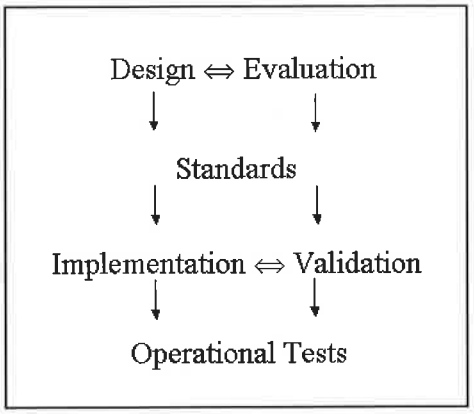

Figure 13: DRBG Assurance Strategy

The design of each DRBG 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 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 by an accredited laboratory (see Section 11.3). The consuming application or cryptographic service that uses a DRBG **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 is correctly implemented. Validation testing for DRBG processes consists of testing whether or not the DRBG process 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 **shall** be implemented within a DRBG 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

This Standard requires the development of a set of documentation that will provide assurance to users and (optionally) validators that the DRBGs 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.
- Document how the implementation has been designed to permit implementation validation and health testing.
- Document the type of DRBG (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 implemention (e.g., prediction resistance, the available elliptic curves, etc.).
- In the case of the **CTR_DRBG**, indicate whether a derivation function is provided. If a derivation function is not used, documentation **shall** clearly indicate that the implementation can only be used if either of the following is available:
    a. An Approved RBG with a security strength equal to or greater than the required security strength of the **CTR_DRBG** instantiation, or
    b. An Approved conditioned entropy source.
- Document any support functions other than health testing.

11.3 Implementation Validation Testing

A DRBG process may be tested for conformance to this Standard. Regardless of whether or not validation testing is obtained by an implementer, a DRBG **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 in derived test requirements.

11.4 Health Testing

11.4.1 Overview

A DRBG implementation **shall** perform self-tests to ensure that the DRBG continues to function properly. Self-tests of the DRBG processes **shall** be performed as specified in Section 9.6. 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 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.

When a DRBG fails a self-test, 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 no pseudorandom bits **shall** be output when an error state exists. When in an error state, user intervention (e.g., power cycling, restart of the DRBG) **shall** be required to exit the error state (see Section 9.7).

11.4.2 Known-Answer Testing

Known-answer testing **shall** be conducted as specified in Section 9.6. A known-answer test involves operating the DRBG 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 **shall** enter an error state and output an error indicator (see Section 9.7).

The generalized known-answer testing is specified in Section 9.6. Testing **shall** be performed on all DRBG functions implemented.

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 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 [8].

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 = 115792089210356248762697446949407573530086 14\
    34152903141955336313088670978539 51

r = 115792089210356248762697446949407573529996 95\
    52241357603424222590610685120443 69

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


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

Py = 4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16 2bce3357 6b315ece
     cbb64068 37bf51f5
```

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

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

A.1.1.2 Curve P-384

```
p = 39402006196394479212279040100143613805079739\
    27046544666794829340424572177149687032904726\
    60882589380018616069731123 19

r = 39402006196394479212279040100143613805079739\
    27046544666794690527962765939911326356939895\
    63081522949135544336539426 43

b = b3312fa7 e23ee7e4 988e056b e3f82d19 181d9c6e fe814112 0314088f
    5013875a c656398d 8a2ed19d 2a85c8ed d3ec2aef

Px = aa87ca22 be8b0537 8eb1c71e f320ad74 6e1d3b62 8ba79b98
     59f741e0 82542a38 5502f25d bf55296c 3a545e38 72760ab7

Py = 3617de4a 96262c6f 5d9e98bf 9292dc29 f8f41dbd 289a147c
     e9da3113 b5f0b8c0 0a60b1ce 1d7e819d 7a431d7c 90ea0e5f

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

Qy = 023b1660 dd701d08 39fd45ee c36f9ee7 b32e13b3 15dc0261
     0aa1b636 e346df67 1f790f84 c5e09b05 674dbb7e 45c803dd
```

A.1.1.3 Curve P-521

```
p = 68647976601306097149819007990813932172694353\
    00143305409394463459185543183397656052122559\
    64066145455497729631139148085803712198799971\
    66438125740282911150571 51

r = 68647976601306097149819007990813932172694353\
    00143305409394463459185543183397655394245057\
    74633321719753296399637136332111386476861244\
    03803403728088927070054 49
```

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

Px = c6858e06 b70404e9 cd9e3ecb 662395b4 429c6481 39053fb5
     21f828af 606b4d3d baa14b5e 77efe759 28fe1dc1 27a2ffa8
     de3348b3 c1856a42 9bf97e7e 31c2e5bd 66

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

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

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

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 point P **shall** remain the generator point G given in Annex A.1 for the selected curve. (This minor restriction simplifies the test procedure to verify just one point each time.)

The point Q **shall** be generated using the procedure specified in ANS X9.62. The following input is required:

> An elliptic curve $E = (F_q, a, b)$, cofactor h, prime n, a bit string *SEED*, and hash function **Hash**(). The curve parameters are given in Annex A. The minimum length m of *SEED* **shall** conform to Section 10.4.1, Table 4, under "Seed length". The bit length of *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 *SEED* must be used.

Otherwise, the output point **shall** be used as the point Q.

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

To insure that the point Q has been generated appropriately, an additional self-test procedure **shall** be performed whenever the instantiate function is invoked. Section 9.6.2 specifies that known-answer tests on the instantiate function be performed prior to creating an operational instantiation. As part of those tests, an implementation of the generation procedure specified in ANS X9.62 **shall** be called with the *SEED* value used to generate the alternate Q. The point returned **shall** be compared with the stored value of Q used in place of the default value (see Annex A.1). 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

Input:

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

Output:

1. x The requested integer representation of the bitstring.

Process:

1. Let $(b_1, b_2, \ldots, b_n)$ 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

Input:

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

Output:

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

Process:

1. Let $(b_1, b_2, \ldots, b_n)$ 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 $b_1, b_2, \ldots, b_n$.

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 an Octet String

Input:

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

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

Output:

1. An octet string O of length n octets.

Process:

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

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

   for $i = 1$ *to* n.
3. Return O.

B.4 Octet String to an Integer

Input:

1. An octet string O of length n octets.

Output:

1. A non-negative integer x.

Process:

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

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

   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, the rth bit of x depends solely upon whether $\left\lfloor \frac{x}{2^r} \right\rfloor$ is odd or even. If all of the values in this range are sampled uniformly, the rth bit will be 0 exactly ½ of the time. 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 $p, p+1, ..., 2^N - 1$ from the set of all integers in Z_{2^N}, the ratio of ones and zeroes in the rth 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 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 a mod p curve with $m = 256$ 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-th bit is kept "secret". About ½ the time, the unpublished bit could easily be determined from the other 255 bits. Similarly, if 254 of the bits are published, about ¼ 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 $2 * p^{½}$ 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.

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 $d << m$.

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

The term in braces represents the approximate number of (*m-d*)-bitstrings that fall into one of $1+2^d$ categories as determined by the number of times j it occurs in an x coordinate; $z = (2f-1)/2f$ is the probability that any particular string occurs in an x coordinate; $p_j = (j*2f)/2^m$ is the probability that a member of the j-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:

log2(*f*): 0 *d*: 0 entropy: 255.00000000 *m-d*: 256

log2(*f*): 0 *d*: 1 entropy: 254.50000000 *m-d*: 255

log2(*f*): 0 *d*: 2 entropy: 253.78063906 *m-d*: 254

log2(*f*): 0 *d*: 3 entropy: 252.90244224 *m-d*: 253

log2(*f*): 0 *d*: 4 entropy: 251.95336161 *m-d*: 252

log2(*f*): 0 *d*: 5 entropy: 250.97708960 *m-d*: 251

log2(*f*): 0 *d*: 6 entropy: 249.98863897 *m-d*: 250

log2(*f*): 0 *d*: 7 entropy: 248.99434222 *m-d*: 249

log2(*f*): 0 *d*: 8 entropy: 247.99717670 *m-d*: 248

log2(*f*): 0 *d*: 9 entropy: 246.99858974 *m-d*: 247

log2(*f*): 0 *d*: 10 entropy: 245.99929521 *m-d*: 246

log2(*f*): 0 *d*: 11 entropy: 244.99964769 *m-d*: 245

log2(*f*): 0 *d*: 12 entropy: 243.99982387 *m-d*: 244

log2(*f*): 0 *d*: 13 entropy: 242.99991194 *m-d*: 243

log2(f): 0 *d*: 14 entropy: 241.99995597 *m-d*: 242

log2(*f*): 0 *d*: 15 entropy: 240.99997800 *m-d*: 241

log2(*f*): 0 *d*: 16 entropy: 239.99998900 *m-d*: 240

Observations:

a) The table starts where it should, at 1 missing bit;

b) The missing entropy rapidly decreases;

c) For $\log2(f) = 0$, i.e, the mod p curves, 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= *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 $[0,1,...,p-1]$ are valid x-coordinates on an elliptic curve defined over 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).

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 already sluggish 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:

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

where k is some constant derived from the asymptotic estimates given in [9]. 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.

C.2 Reserve for a discussion of the nonce specified in Section 8.4.2, item 7

ANNEX D: (Informative) DRBG 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 some 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 generated, and that this must be done with great care, or the cryptographic mechanisms will be weakened. At this point, there are three things that may guide the designer's choice of a DRBG:

a. He may already have decided to include a set of cryptographic primitives as part of his implementation. By choosing a DRBG based on one of these primitives, he can minimize the cost of adding that DRBG. 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 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 based on similar properties, he can minimize the number of algorithms he has to trust.

   For example, an AES-based DRBG might be a good choice when a module provides encryption with AES. Since the security of the DRBG 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).

The DRBGs 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 DRBG computes

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

At the end of a generation request, the DRBG 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, though unproven, assumption 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 10.2.2.2.1).

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 (see ASC X9 Registry 00002). At the present time, only three-key TDEA and AES are approved for use in this DRBG. 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.

Comment [ebb2]: This reference is only useful for AES, not TDEA.

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 parallelizeable. 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, 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 <u>not</u> 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 1) an Approved RBG with a security strength equal to or greater than the required security strength of the **CTR_DRBG** instantiation, or 2) an Approved conditioned entropy source.

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, this algorithm generates pseudorandom bits more slowly than the other DRBGS 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 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. A given implementation of this DRBG 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, whether for two million bits or a single bit, 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 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 is a natural choice, as it allows the designer to utilize existing capabilities to generate truly high-security random numbers.

Security. The security of **Dual_EC_DRBG** is based on the so-called "Elliptic Curve Discrete Logarithm Problem" that has no known attacks better than the so-called "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 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, 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 entropy input source 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 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 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.

ANNEX E: (Informative) Example Pseudocode for Each DRBG

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, arbitary values have been selected that are consistent with the allowed values for each DRBG, 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* ≠ "Success", an invalid *state_handle* is returned.

When the uninstantantiate 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 provided. 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 **Update** function is specified in Section 10.2.2.2.2.

A personalization string is allowed, 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 supported. 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 *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.

Instantiate_HMAC_DRBG (...):

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 × *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*) = **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(...):

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*).

Comment: Get the internal state.

2. *V* = *internal_state* (*state_handle*).*V*, *Key* = *internal_state* (*state_handle*).*Key*, *security_strength* = *internal_state* (*state_handle*).*security_strength*, *reseed_counter* = *internal_state (state_handle).reseed_counter*.

Comment: Check the validity of the rest of the input parameters.

3. If (*requested_no_of_bits* > 7500), then **Return** ("Too many bits requested", *Null*).
4. If (*requested_security_strength* > *security_strength*), then **Return** ("Invalid *requested_security_strength*", *Null*).

Comment: Invoke the generate algorithm.

5. (*status*, *pseudorandom_bits*, *V*, *Key*, *reseed_counter*) = **Generate_algorithm** (*V*, *Key*, *reseed_counter*, *requested_number_of_bits*).
6. If (*status* = "Reseed required"), then **Return** ("DRBG can no longer be used. Please re-instantiate or reseed", *Null*).
7. *internal_state* (*state_handle*) = {*V*, *Key*, *security_strength*, *reseed_counter*}.
8. **Return** ("Success", *pseudorandom_bits*).

Generate_algorithm (...):

Input: bitstring (*V*, *Key*), integer (*reseed_counter*, *requested_number_of_bits*).

Output: string *status*, bitstring (*pseudorandom_bits*, *V*, *Key*), integer *reseed_counter*.

Process:

1 If (*reseed_counter* ≥ 10,000), then **Return** ("Reseed required", *Null*, *V*, *Key*, *reseed_counter*).

2. *temp* = *Null*.

3 While (**len** (*temp*) < *requested_no_of_bits*) do:

3.1 *V* = **HMAC** (*Key*, *V*).

3.2 *temp* = *temp* || *V*.

4. *pseudorandom_bits* = Leftmost (*requested_no_of_bits*) of *temp*.
5. (*Key*, *V*) = **Update** (*additional_input*, *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 available, and a block cipher derivation function using AES-128 is used. Both a personalization string and additional input are allowed. A total of 5 internal states are available. For this implementation, the functions and algorithms are written as separate routines. The **Block_Encrypt** function (specified in Section 10.5.3) 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 installed (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 always available, 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. Note that for this value, the instantiation count will not repeat during the reseed interval.

E.3.2 The Update Function

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* || *ouput_block.*
4. *temp* = Leftmost 256 bits of *temp.*
5 *temp* = *temp* ⊕ *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 always available.

Instantiate_CTR_DRBG (...):

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* ≤ 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*) = **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. *internal_state_* (*state_handle*) = {*V*, *Key*, *reseed_counter*, *security_strength*}.
11. **Return** ("Success", *state_handle*).

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*) = **Update** (*seed_material*, *Key*, *V*).
6. *reseed_counter* = 1.
7. **Return** (*V*, *Key*, *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.

Reseed_CTR_DRBG_Instantiation (...):

Input: integer (*state_handle*), bitstring *additional_input*.

Output: string *status*.

Process:

Comment: Check for the validity of *state_handle*.

1. If ((*state_handle* < 0) or (*state_handle* > 4) or (*internal_state*(*state_handle*) = {*Null*, *Null*, 0, 0}), then **Return** ("State not available for the indicated *state_handle*").

Comment: Get the internal state values.

2. *V* = *internal_state* (*state_handle*).*V*, *Key* = *internal_state* (*state_handle*).*Key*, *security_strength* = *internal_state* (*state_handle*).*security_strength*.
3. If (**len** (*additional_input*) > 800), then **Return** ("*Additional_input* too long").
4. (*status*, *entropy_input*) = **Get_entropy_input** (*security_strength*, *security_strength*, 1000).
6. If (*status* ≠ "Success"), then **Return** ("Catastrophic failure of the entropy source:" || *status*).

Comment: Invoke the reseed algorithm.

7. (*V*, *Key*, *reseed_counter*) = **Reseed_algorithm** (*V*, *Key*, *reseed_counter*, *entropy_input*, *additional_input*).
8. *internal_state* (*state_handle*) = {*V*, *Key*, *reseed_counter*, *security_strength* }.
9. **Return** ("Success").

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* || *additional_input*.
2. *seed_material* = **Block_Cipher_df** (*seed_material*, 256).
3. (*Key*, *V*) = **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(...):

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) or (*state_handle* > 4) or (*internal_state* (*state_handle*) = {*Null*, *Null*, 0, 0}), then **Return** ("State not available for the indicated *state_handle*", *Null*).

Comment: Get the internal state.

2. *V* = *internal_state* (*state_handle*).*V*, *Key* = *internal_state* (*state_handle*).*Key*, *security_strength* = *internal_state* (*state_handle*).*security_strength*, *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* = **Reseed_CTR_DRBG_Instantiation** (*state_handle*, *additional_input*).

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

Comment: Get the new working state values; the administrative information was not affected.

7.3 *V* = *internal_state* (*state_handle*).*V*, *Key* = *internal_state* (*state_handle*).*Key*, *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*) = **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. *internal_state* (*state_handle*) = {*V*, *Key*, *reseed_counter*, *security_strength*).

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

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

3. *temp* = *Null*.

4. While (**len** (*temp*) < *requested_number_of_bits*) do:

   4.1 $V = (V + 1) \bmod 2^{128}$.

   4.2 *output_block* = **AES_ECB_Encrypt** (*Key*, *V*).

   4.3 *temp* = *temp* || *ouput_block*.

5. *returned_bits* = Leftmost (*requested_number_of_bits*) of *temp*.

6. *zeros* = 0^{256}. Comment: Produce a string of 256 zeros.

7. (*Key*, *V*) = **Update** (*zeros*, *Key*, *V*)

8. *reseed_counter* = *reseed_counter* + 1.

9. **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 allowed. A total of 5 internal states are available. For this implementation, the functions and algorithms are written as separate routines. The **Block_Encrypt** function (as specified in Section 10.5.4) uses AES-128 in the ECB mode.

The nonce for instantiation (*instantiation_nonce*) consists of a 32-bit incrementing counter that is the initial 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 installed (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 always available, 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 ≤ *seedlen,* where *seedlen* = 256 bits). In addition, the maximum size of any *additional_input* is 256 bits (i.e., len (*additional_input* ≤ *seedlen*)).

E.4.2 The 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 (**Instantiate_CTR_DRBG**) 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* || *personalization_string*.

The instantiate algorithm (**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* || $0^{256-temp}$.

seed_material = *entropy_input* ⊕ *personalization_string*.

E.4.4 Reseeding a CTR_DRBG Instantiation Without a Derivation Function

The reseed function (**Reseed_CTR_DRBG**) 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 (**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* || $0^{256-temp}$.

seed_material = *entropy_input* ⊕ *additional_input*.

E.4.5 Generating Pseudorandom Bits Using CTR_DRBG

The generate function (**CTR_DRBG**) 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 (**Generate_algorithm**) is the same as that provided in Annex E.3.5, except that step 2.1 is replaced by:

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

If (*temp* < 256), then *additional_input* = *additional_input* || $0^{256-temp}$.

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:

requested_instantiation_security_strength	**Elliptic Curve**
≤ 112	P-256
113 – 128	P-256
129 – 192	P-384
193 – 256	P-521

A reseed capability is available, but prediction resistance is not available. 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 *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*, *block_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. *highest_supported_security_strength* = 256.
2. Output block length (*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 × *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.

Instantiate_Dual_EC_DRBG (...):

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. *block_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. *internal_state* (*state_handle*) = {*s*, *seedlen*, *p*, *a*, *b*, *n*, *P*, *Q*, *block_counter*, *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.

Reseed_Dual_EC_DRBG_Instantiation (...):

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

Output: string *status*.

Process:

Comment: Check the input parameters.

1. If ((*state_handle* < 0) or (*state_handle* > 9) 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").

Comment: Get the appropriate *state* values for the indicated *state_handle*.

3. *s* = *internal_state* (*state_handle*).*s*, *seedlen* = *internal_state* (*state_handle*).*seedlen*, *security_strength* = *internal_state* (*state_handle*).*security_strength*.

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

3. (*status*, *entropy_input*) = **Get_entropy_input** (*security_strength*, 1000).
4. If (*status* ≠ "Success"), then **Return** ("Catastrophic failure of the entropy source:"|| *status*).

Comment: Perform the reseed algorithm.

5. *seed_material* = **pad8** (*s*) || *entropy_input* || *additional_input*.
6. *s* = **Hash_df** (*seed_material*, *seedlen*).

Comment: Update the changed values in the *state*.

7. *internal_state* (*state_handle*).*s* = *s*.
8. *internal_state*.*block_counter* = 0.
9. **Return** ("Success").

E.5.4 Generating Pseudorandom Bits Using Dual_EC_DRBG

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

Dual_EC_DRBG (...):

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) or (*state_handle* > 9) or (*internal_state* (*state_handle*) = 0)), then **Return** ("State not available for the *state_handle*", *Null*).

Comment: Get the appropriate *state* values for the indicated *state_handle*.

2. *s* = *internal_state* (*state_handle*).*s*, *seedlen* = *internal_state* (*state_handle*).*seedlen*, *P* = *internal_state* (*state_handle*).*P*, *Q* = *internal_state* (*state_handle*).*Q*, *block_counter* = *internal_state* (*state_handle*).*block_counter*.

Comment: Check the rest of the input parameters.

3. If (*requested_number_of_bits* > 1000), then **Return** ("Too many bits requested", *Null*).
4. If (*requested_security_strength* > *security_strength*), then **Return** ("Invalid *requested_strength*", *Null*).
5. If (**len** (*additional_input*) > 800), then **Return** ("Additional_input too long", *Null*).

Comment: Check whether a reseed is required.

6. If $\left(block_counter + \left\lceil \frac{requested_number_of_bits}{outlen} \right\rceil > 2^{32}\right)$, then

   6.1 **Reseed_Dual_EC_DRBG_Instantiation** (*state_handle*, *additional_input*).

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

   6.3 *s* = *internal_state* (*state_handle*).*s*, *block_counter* = *internal_state* (*state_handle*).*block_counter*.

6.4 *additional_input* = *Null*.

Comment: Execute the generate algorithm.

7. If (*additional_input* = *Null*) then *additional_input* = 0

Comment: *additional_input* set to *m* zeroes.

Else *additional_input* = **Hash_df** (**pad8** (*additional_input*), *seedlen*).

Comment: Produce *requested_no_of_bits*, *outlen* bits at a time:

8. *temp* = the *Null* string.
9. $i = 0$.
10. $t = s \oplus additional_input$.
11. $s = \varphi(x(t * P))$.
12. $r = \varphi(x(s * Q))$.
13. *temp* = *temp* || (**rightmost** *outlen* bits of r).
14. $additional_input = 0^{seedlen}$. Comment: *seedlen* zeroes; *additional_input* is added only on the first iteration.
15. *block_counter* = *block_counter* + 1.
16. $i = i + 1$.
17. If (**len** (*temp*) < *requested_no_of_bits*), then go to step 10.
18. *pseudorandom_bits* = **Truncate** (*temp*, $i \times outlen$, *requested_no_of_bits*).

Comment: Update the changed values in the *state*.

19. *internal_state.s* = *s*.
20. *internal_state.block_counter* = *block_counter*.
21. **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 DRBGs 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 levels may be supported in Part 3 by requesting the appropriate security level during instantiation and generation (see Sections 8.2.4, 9.2 and 9.4), and by the use of an appropriate entropy input source (see Parts 2 and Parts 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). Parts 2 and 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 DRBGs 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 level 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 DRBGs in this Standard have been designed to provide these properties when provided with sufficient entropy as discussed in Parts 2 and 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 DRBGs 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 level. 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 function (see Sections 9.2, 9.3 and 9.4), the tests to be made during health testing (see Section 9.6) 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 boundary to provide this property. Requirements for the DRBG 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 is discussed in Sections 9.6 and 11.4.

8. *An RBG is designed to support backtracking resistance.*

   The DRBGs in Part 3 have been designed to support backtracking resistance (see Section 8.6).

9. *An RBG may support prediction resistance.*

   A DRBG may be designed and implemented to support prediction resistance. See Annex F.4 for additional information.

ANNEX G: (Informative) Bibliography

[1] Handbook of Applied Cryptography; Menezes, van Oorschot and Vanstone; CRC Press, 1997

[2] Applied Cryptography, Schneier, John Wiley & Sons, 1996

[3] RFC 1750, Randomness Recommendations for Security, IETF Network Working Group; Eastlake, Crocker and Schiller; December 1994.

[4] Cryptographic Random Numbers, Ellison, submission for IEEE P1363.

[5] Cryptographic Randomness from Air Turbulence in Disk Drives; Davis, Ihaka and Fenstermacher.

[6] Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator; Kelsey, Schneier, and Ferguson.

[7] The Intel® Random Number Generator; Cryptography Research, Inc.; White paper prepared for Intel Corporation; Jun and Kocher; April 22, 1999.

[8] Federal Information Processing Standard 186-3, *Digital Signature Standard (DSS)*, [Date to be inserted].

[9] [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.qu.