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

August 2005

Table of Contents

Random Number Generation
Part 3: Deterministic Random Bit Generator Mechanisms

1 Scope

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.

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

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 E for a discussion of DRBG selection.

2 Conformance

An implementation of a deterministic random bit generator (DRBG) may claim conformance with ANSI 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 document. 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 X9.62-2000, *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 180-2, *Secure Hash Standard (SHS)*, August 2002; ASC X9 Registry 00003.

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

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

4 Terms and definitions

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

5 Symbols

The following symbols are used in this document.

Symbol	Meaning
+	Addition
$\lceil X \rceil$	Ceiling: the smallest integer $\geq X$. For example, $\lceil 5 \rceil = 5$, and $\lceil 5.3 \rceil = 6$.
$X \oplus Y$	Bitwise exclusive-or (also bitwise addition mod 2) of two bitstrings X and Y of the same length.

$X \| Y$	Concatenation of two strings X and Y. X and Y are either both bitstrings, or both 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.
$\{a_1, ...a_i\}$	The internal state of the DRBG at a point in time. The types and number of the a_i depends on the specific DRBG.
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, specifies the characteristics for random numbers and random number generators, and provides mathematical and cryptographic background information on the concept of randomness. Random bit generators are used for the generation of random numbers. Part 1 specifies requirements for random bit generators that are applicable to both non-deterministic random bit generators (NRBGs) and deterministic random bit generators (DRBGs). In addition, Part 1 also introduces a general functional model and a conceptual cryptographic Application Programming Interface (API) for random bit 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 seed and reseed the DRBG.

Part 4 of this Standard (*Random Bit Generator Constructions*) provides guidance on combining components to construct 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. 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 seed and any other input information. However, if the input is kept secret, and the algorithm is well designed, the bitstrings will appear to be random.

Comment [ebb1]: Page: 21
Mike to provide alternate text ?

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 goals 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 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.

— Annex C discusses security considerations for selecting and implementing DRBGs.

The following informative annexes are also included:

— Annex D discusses the functional requirements specified in Part 1 as they are fulfilled by this part of the Standard.

— Annex E provides a discussion on DRBG selection.

— Annex F provides example pseudocode for each DRBG.

— 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.

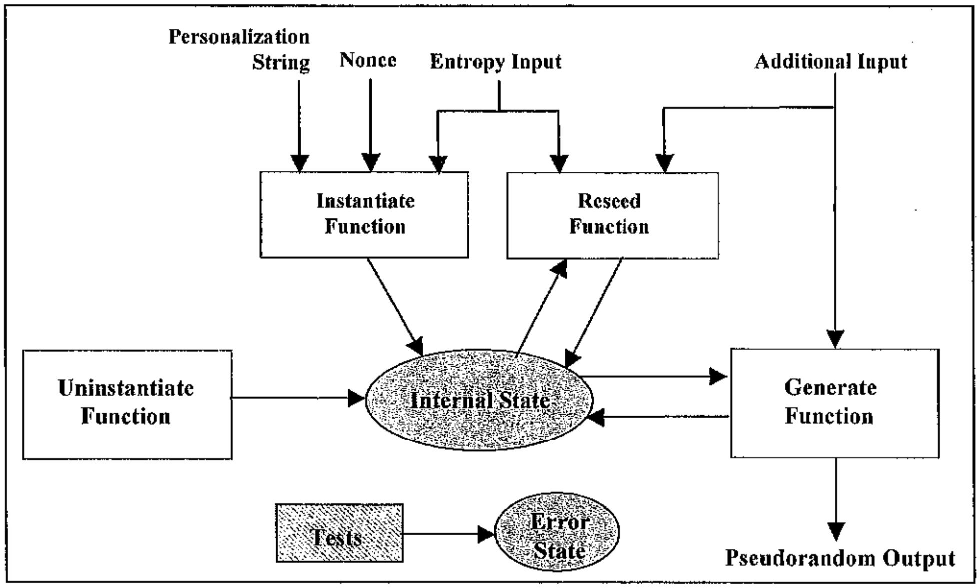

Figure 1: DRBG Functional Model

7.2 Functional Model Components

7.2.1 Introduction

Part 1 of this Standard provides general functional requirements for random bit generators. These requirements are discussed briefly in this section.

Comment [ebb2]: Page: 23
Does the material in Annex D need to be included here ?

7.2.2 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.

The DRBGs, as specified in this part of the Standard and further discussed in Part 4, allow for some bias in the entropy input. Whenever a bitstring containing entropy is required by the DRBG, a request is made that indicates the minimum amount of entropy to be returned; the request may obtain entopy input bits from a buffer containing readily available entopy bits or may cause entropy input bits to be acquired. The request may be fulfilled by a bitsting that is equal to or greater in length than the requested entropy. The DRBG expects that the returned bitstring will contain at least the amount of entropy requested. Additional entropy beyond the amount requested is not required, but is desirable.

7.2.3 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 is required and is combined with the entropy input to create the initial DRBG seed. Criteria for the nonce are provided in Section 8.4.

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.

7.2.4 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 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.

7.2.5 The DRBG Functions

The DRBG functions handle the DRBG's internal state. The DRBGs in this Standard have four separate functions:

1. The instantiate function acquires entropy input and combines 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.

7.2.6 Testing

Testing is concerned with assessing and reacting to the health of the DRBG. The health tests are discussed in Sections 9.7 and 11.4.

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 testing functions. A DRBG **may** also include a reseed function. A DRBG **shall** be instantiated prior to the generation of output by the DRBG.

8.2.2 DRBG Instantiations

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

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

8.2.3 Internal States

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

1. Working state:
    a. One or more values that are derived from the seed and become part of the internal state; these values must usually remain secret, and
    b. A count of the number of requests or blocks produced since the instantiation

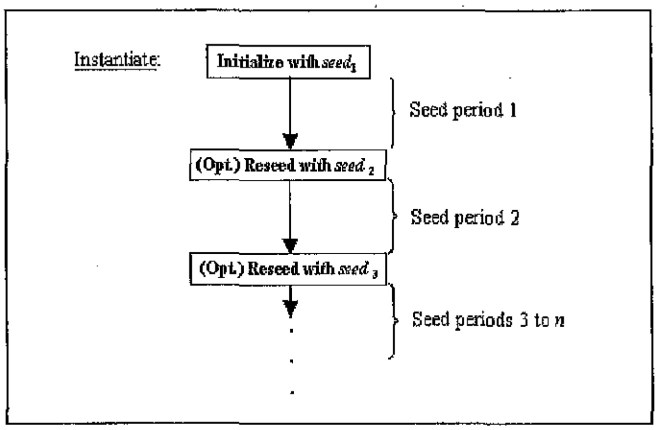

Figure 2: DRBG Instantiation

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 **may** 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 may be instantiated to support only a 128-bit security strength.

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 level (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 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 distibuted 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.

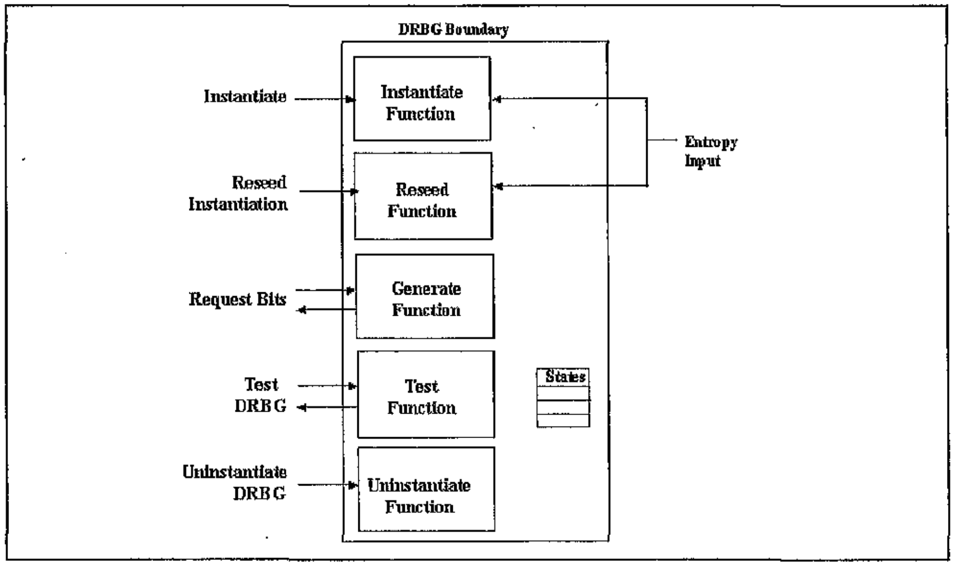

Figure 3: DRBG Functions Within a Single Device

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.

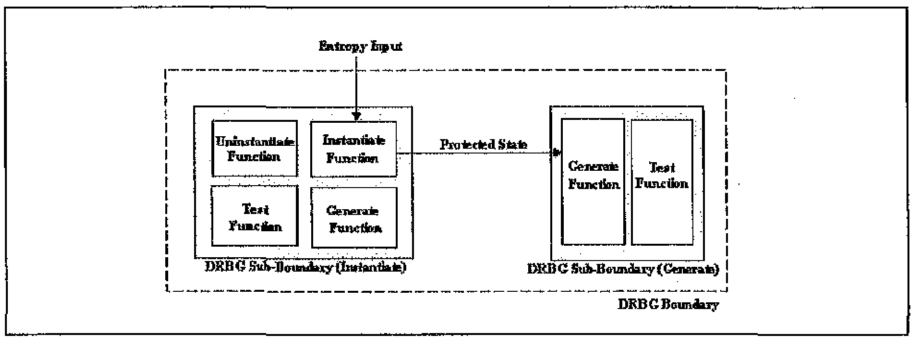

Figure 4: Distributed DRBG Functions

Each DRBG boundary or sub-boundary **shall** contain an uninstantiate function and a test function to test the "health" of other DRBG functions within that boundary.

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 SP 800-57).

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 recovering the secrecy of the output of the DRBG if a seed or the internal state becomes known. Periodic reseeding is a good countermeasure to the potential threat that the seeds and DRBG output become compromised. 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 be used in the construction of a seed; requirements for the entropy input are discussed in item 3. A nonce is also be 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.

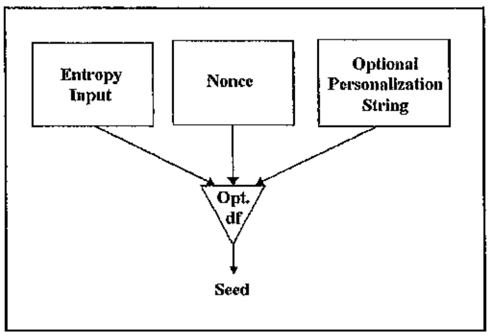

Figure 5: Seed Construction for Instantiation

Depending on the DRBG and the source of the entropy input, a derivation function is required to derive a seed from the seed material. When full entropy input is readily available, the DRBGs based on block cipher algorithms (see Section 10.2) may be implemented without a derivation function. When implemented in this manner, a nonce is not used as shown in Figure 5. Note, however, that the personalization string could contain a nonce, if desired.

The goal of this seed construction is to ensure that the seed is statistically unique.

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, optonally, 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.

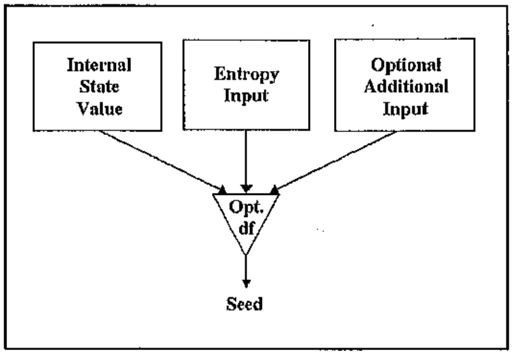

Figure 6: Seed Construction for Reseeding

3. Entropy requirements for the entropy input: The entropy input for the seed **shall** contain sufficient entropy for the desired security strength. Additional entropy **may** be provided in the nonce or the optional personalization string during instantiation, or in the additional input during reseeding, but this is not required. Entropy contained in the seed components is distributed across the seed (e.g., using an appropriate derivation function) by the instantiate and reseed functions.

   The entropy input **shall** have entropy that is equal to or greater than the security strength of the instantiation. Note that 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.

Comment [ebb3]: Page: 31
This may need to be revised if the Dual_EC_DRBG is not retained.

5. Entropy input source: The source of the entropy input **may** be an Approved NRBG, an Approved DRBG (or chain of Approved DRBGs) that is seeded by an Approved NRBG, or an Approved entropy source. Further discussion about the entropy input is provided in Parts 2 and 4 of this Standard.

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

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** be treated at least as well as the key.

7. Nonce: A nonce is required to construct a seed during instantation. 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 guaranteed to never repeat, or

   c. 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.

8. Reseeding: Generating too many outputs from a seed (and other input information) may provide sufficient information for successfully predicting future outputs unless prediction resistance is provided (see Section 8.6). 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 application, or 2) by the generate function when prediction resistance is requested (see Section 8.6) or the limit of the seedlife is reached. An alternative to reseeding is to create an entirely new instantiation.

   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 old seed and newly-obtained entropy input that will support the desired security strength.

9. Seed use: DRBGs **may** be used to generate both secret and public information. In either case, the seed and the entropy input from which the seed is derived **shall** be kept secret. A single instantiation of a DRBG **should not** be used to generate both secret and public values. However, cost and risk factors must be taken into account when determining whether different instantiations for secret and public values can be accommodated.

   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.

Comment [EBB4]: Page: 32
Should this be addressed in Part 4 ?

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

10. Seed separation: Seeds used by DRBGs **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 8.4, item 1). 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 application. For example, the input could be derived directly from values entered by the user or application, or the input could be derived from information introduced by the user or application (e.g., from timing statistics based on key strokes), 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). The intent of a personalization string is to differentiate this DRBG instantiation from all the others that might ever appear. 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:

1. Device serial numbers,
2. Public keys,
3. User identification,
4. Private keys,
5. PINs and passwords,
6. Secret per-module or per-device values,
7. Timestamps,
8. Network addresses,
9. Special secret key values for this specific DRBG instantiation,

10. Application identifiers,
11. Protocol version identifiers,
12. Random numbers, and
13. 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 may be either secret or publicly known; its value is arbitrary, although its length 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 seed or one or more DRBG internal states.

8.6 Prediction Resistance and Backtracking Resistance

Figure 7 depicts the sequence of DRBG internal states that result from a given seed. The internal state is used to generate pseudorandom bits upon request by a user. The following discussions will use the figure to explain backtracking and prediction resistance. Suppose that a compromise occurs at $State_x$, where $State_x$ contains both secret and public information.

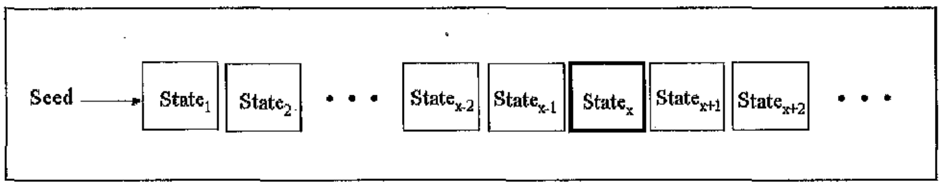

Figure 7: Sequence of DRBG States

Backtracking Resistance: Backtracking resistance means that a compromise of the DRBG internal state has no effect on the security of prior outputs. That is, an adversary who is given access to all of ~~any subset of~~ that prior output sequence cannot distinguish it from random; if the adversary knows only part of the prior output, he cannot determine any bit of that prior output sequence that ~~the adversary~~he has not already seen. ~~In other words, a compromise has no effect on the security of prior outputs.~~

For example, suppose that an adversary knows $State_x$ ~~and also knows the output bits from $State_1$ to $State_{x-2}$~~. Backtracking resistance means that:

a. The output bits from $State_1$ to $State_{x-1}$ cannot be distinguished from random.

~~a.~~ b. The prior internal state values themselves ($State_1$ to $State_{x-1}$) cannot be

Formatted: Bullets and Numbering

Formatted

Formatted

recovered, given knowledge of the secret information in $State_x$. ~~$State_{x-1}$ and its output bits cannot be determined from knowledge of $State_x$ (i.e., $State_x$ cannot be "backed up"). In addition, since the output bits from $State_1$ to $State_{x-2}$ appear to be random, the output bits for $State_{x-1}$ cannot be predicted from the output bits of $State_1$ to $State_{x-2}$.~~

Comment [ebb5]: Page: 34
This makes the definition very convoluted.

Backtracking resistance can be provided by ensuring that the internal state transition function of a DRBG is a one-way function. All DRBGs in this Standard have been designed to provide backtracking resistance.

Prediction Resistance: Prediction resistance means that a compromise of the DRBG internal state has no effect on the security of future DRBG outputs. ~~If a compromise of $State_x$ occurs, *prediction resistance* provides assurance that the output sequence resulting from states *after* the compromise remains secure.~~ That is, an adversary who is given access to all of ~~any subset of~~ the output sequence after the compromise cannot distinguish it from random; if the adversary knows only part of the future output sequence, ~~an adversary~~he cannot predict any bit of that future output sequence that he has not already seen. ~~In other words, *a compromise has no effect on the security of future outputs.*~~

For example, suppose that an adversary knows $State_x$: ~~and also knows the output bits from $State_{x+2}$ to $State_{x+n}$.~~ Prediction resistance means that:

a. The output bits from $State_{x-1}$ and forward cannot be distinguished from an ideal random bitstring by the adversary.

Formatted: Bullets and Numbering

Formatted

~~b.~~ b. The future internal state values themselves ($State_{x+1}$ and forward) cannot be predicted, given knowledge of $State_x$. ~~$State_{x-1}$ and its output bits cannot be determined from knowledge of $State_x$ (i.e., $State_x$ cannot be "backed up"). In addition, since the output bits from $State_1$ to $State_{x-2}$ appear to be random, the output bits for $State_{x-1}$ cannot be predicted from the output bits of $State_1$ to $State_{x-2}$.~~

Formatted

~~$State_{x+1}$ and its output bits cannot be predicted from knowledge of $State_x$. In addition, because the output bits from $State_{x+2}$ to $State_{x+n}$ appear to be random, the output bits for $State_{x+1}$ cannot be determined from the output bits of $State_{x+2}$ to $State_{x+n}$.~~

Prediction resistance can be provided only by ensuring that a DRBG is effectively reseeded between DRBG requests. That is, an amount of entropy that is sufficient to support the security strength of the DRBG (i.e., an amount that is at least equal to the security strength) must be added to the DRBG in a way that ensures that knowledge of the current~~previous~~ DRBG internal state does not allow an adversary any useful knowledge about future DRBG internal states or outputs.

9 DRBG Functions

9.1 General Discussion

The DRBG functions in this Standard are specified as an algorithm and an "envelope" of pseudocode around that algorithm. 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, but the function **shall** have equivalent functionality.

In the specifications of this Standard, the following pseudo-functions are used. These functions are not specifically defined in this Standard, but have the following meaning:

- **Get_entropy**: A function that is used to obtain entropy input. The function call is:

  (*status*, *entropy_input*) = **Get_entropy** (*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.

- **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; for AES, the basic encryption operation is called the cipher operation. The basic encryption operation is equivalent to an encryption operation on a single block of data using the ECB mode.

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** 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 **shall**:

1. Check the validity of the other input parameters,
2. Determine the security strength for the DRBG instantiation,
3. Determine any DRBG specific parameters (e.g., elliptic curve domain parameters),
4. Obtain entropy input with entropy sufficient to support the security strength,
5. Obtain the nonce,

6. Determine the initial internal state using the instantiate algorithm,
7. If possible, request that pseudorandom bits be generated; the generate function will test that successive internal state values are not identical.
8. Return a *state_handle* for the internal state to the consuming application.

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). If a generate function is not contained in the same sub-boundary as the instantiate function, steps 13 and 14 are not performed.

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

Input from a consuming application:

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 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 12.
3. *personalization_string*: An optional input that provides personalization information (see Sections 8.4 and 8.5.2). The maximum length of the personalization string (*max_personalization_string_length*) is implementation dependent, but **shall** be $\leq 2^{35}$ bits. If a personalization string will never be used, then the input parameter and step 3 may be omitted, and step 9 may be modified to omit the personalization string.
4. *DRBG_specific_input_parameters* : Any additional parameters that are allowed for a specific DRBG (see Section 10). The use of the DRBG-specific input parameters is discussed for the DRBG instantiate algorithms. If a DRBG or a DRBG implementation does not use these parameters, then step 5 may be omitted.

Required information not provided by the consuming application:

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 $\leq 2^{35}$ bits.

2. *nonce*: A nonce as specified in Section 8.4. Note that if a random value is used as the nonce, the *entropy_input* and *nonce* could be acquired using a single **Get_entropy** call (see step 6); 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), step 8 would be omitted, and the *nonce* would be omitted from the parameter list in step 9.

Output to a consuming application:

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:

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

Process:

Comment: Check the validity of the input parameters.

1. If *requested_instantiation_security_strength* > *highest_supported_security_strength*, then reruen an **ERROR**.

2. If *prediction_resistance_flag* is set, and prediction resistance is not supported, then return an **ERROR**.

3. If the length of the *personalization_string* > *max_personalization_string_length*, return an **ERROR**.

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 10.3.2.2.2), and by the MS_DRBG (see Section 10.3.3.2.3). Otherwise, the step should be omitted.

5. Using *security_strength* and *DRBG_specific_input_parameters* (if available), select appropriate DRBG parameters.

Comment: Obtain the entropy input.

6. (*status, entropy_input*) = **Get_entropy** (*security_strength*, *min_length*, *max_length*).

7. If an **ERROR** is returned in step 6, return an **ERROR**.

8. Obtain a *nonce*. Comment: This step **shall** include any appropriate checks on the acceptability of the *nonce*. See Section 8.4

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

9. (*status*, *working_state*) = **Instantiate_algorithm** (*entropy_input*, *nonce*, *personalization_string*, *other DRBG_parameters*).

10. If an ERROR is returned from step 9, then

    10.1 Delete all instantiations using the uninstantiate function.

    10.2 Return the **ERROR** status from step 9.

    Comment: Set up the initial internal state.

11. Get a *state_handle* that will be used to locate the internal state for this instantiation. If an unused internal state cannot be found, return an **ERROR**.

12. Set the internal state indicated by *state_handle* to the initial values for the *working_state* and administrative information, as appropriate.

    Comment: Invoke the generate function in Section 9.4 to test that two consecutive internal states are not identical[2]. Ignore the returned pseudorandom bits.

13. (*status*, *pseudorandom_bits*) = **Generate_Function** (*state_handle*, 64, *security_strength*, *No_prediction_resistance*, *Null*, *additional_input*).

14. If *status* indicates that two consecutive internal states were identical, then

    14.1 Delete all instantiations using the uninstantiate function.

    14.2 Return the **ERROR** status from step 14.

15. Return **SUCCESS** and *state_handle*.

9.3 Reseeding a DRBG Instantiation

The reseeding of an instantiation is not required, but is recommended whenever an 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 an application,

[2] This is the continuous random number test from FIPS 140-2.

- performed when prediction resistance is requested by an application,
- triggered by the generate function when a predetermined number of pseudorandom outputs have been produced (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 **shall**:

1. Check the validity of the input parameters,
2. Obtain entropy input with sufficient entropy to support the security strength, and
3. Using the reseed algorithm, combine the current working state with the new entropy input and any additional input to determine the new working state. The reseed algorithm will check that two consecutive states are different.

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:

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 $\leq 2^{35}$ bits. If *additional_input* will never be used, then the input parameter and step 2 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:

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 $\leq 2^{35}$ bits.
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:

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

Information retained within the DRBG boundary:

Replaced internal state values (i.e., the *working_state*).

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 unused internal state, return an **ERROR**.
2. If the length of the *additional_input* > *max_additional_input_length*, return an **ERROR**.

Comment: Obtain the entropy input.

3. (*status*, *entropy_input*) = **Get_entropy** (*security_strength*, *min_length*, *max_length*).
4. If an **ERROR** is returned in step 3, return an **ERROR**.

Comment: Get the new *working_state* using the appropriate reseed algorithm in Section 10.

5. (*status*, *working_state*) = **Reseed_algorithm** (*working_state*, *entropy_input*, *additional_input*).

Comment: If an **ERROR** is returned, two consecutive states are the same.

6. If an **ERROR** is returned from step 6, then
   6.1 Delete all instantiations using the uninstantiate function.
   6.2 Return the **ERROR** *status* from step 5.

Comment: Save the new values of the internal state.

7 Replace the *working_state* in the internal state indicated by *state_handle* with the new values.

8. 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 **shall**:

1. Check the validity of the input parameters,
2. If the instantiation needs additional entropy because the end of the seedlife has been reached or prediction resistance is required, call the reseed function to obtain

sufficient entropy.

3. Generate the requested pseudorandom bits using the generate algorithm. The generate algorithm will check that two consecutive states are not the same.

4. Update the working state.

5. Return the requested pseudorandom bits to the consuming appication.

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:

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 ≤ 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 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 the *prediction_resistance_request* parameter is not needed, then the input parameter and step 5 may be omitted.

   If prediction resistance is never provided, then step 5 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 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 **ERROR**.

   Using *state_handle*, obtain the new internal state.

   (*status*, *pseudorandom_bits*, *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 $\leq 2^{35}$

bits. If *additional_input* will never be used, then the input parameter, step 4, step 7.4 and the *additional_input* input parameter in step 8 may be omitted.

Required information not provided by the consuming application:

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

Output to a consuming application:

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:

Replaced internal state values (i.e., the *working_state*).

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 unused internal state, then return an **ERROR**.
2. If *requested_number_of_bits* > *max_number_of_bits_per_request*, then return an **ERROR**.
3. If *requested_security_strength* > the *security_strength* indicated in the internal state, then return an **ERROR**.
4. If the length of the *additional_input* > *max_additional_input_length*, then return an **ERROR**.
5. If *prediction_resistance_request* is set, and *prediction_resistance_flag* is not set, then return an **ERROR**.
6. Clear the *reseed_required_flag*.

Comment: Get the requested pseudorandom bits.

7. If *reseed_required_flag* is set, or if *prediction_resistance_request* is set, then

Comment: Reseed the instantiation (see Section 9.3).

   7.1 *status* = **Reseed** (*state_handle*, *additional_input*).

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

   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*, *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.

Comment: If an **ERROR** is returned, two consecutive states are the same.

10. If an **ERROR** is returned from step 8,

    10.1 Delete all instantiations using the uninstantiate function.

    10.2 Return the **ERROR** received from step 8.

10. Replace the old *working_state* in the internal state indicated by *state_handle* with the 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 If an **ERROR** is returned in step 9.1, then return the **ERROR**.

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

9.5 Removing a DRBG Instantiation

The internal state for an instantiation may need to be "released". This may be required, for example, following health testing of the instantiation function. The uninstantiate function **shall**:

1. Check the input parameter for validity.

2. Empty 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:

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

Output to a consuming application:

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

Information retained within the DRBG boundary:

An empty internal state.

Process:

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

9.6 Auxilliary Functions

9.6.1 Introduction

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 9.6.2), and the other method is based on block cipher algorithms (see 9.6.3). The block cipher derivation function uses a a CBC_MAC that is specified in Section 9.6.4.

9.6.2 Derivation Function Using a Hash Function (Hash_df)

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 $\leq (255 \times outlen)$. *no_of_bits_to_return* is represented as a 32-bit integer.

Output:

1. *status*: The status returned from **Hash_df**. The status will indicate **SUCCESS** or

ERROR.

2. *requested_bits* : The result of performing the **Hash_df**.

Process:

1. If *no_of_bits_to_return* > *max_number_of_bits*, then return an **ERROR**.
2. *temp* = the Null string.
3. $len = \left\lceil \frac{no\_of\_bits\_to\_return}{outlen} \right\rceil$.
4. *counter* = a 32-bit binary value representing the integer "1".
5. For *i* = 1 to *len* do

   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*.

9.6.3 Derivation Function Using a Block Cipher Algorithm

Let **Block_Cipher_Hash** be the function specified in Section 9.6.4. Let Let *outlen* be its output block length, 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**.
2. *requested_bits* : The result of performing the **Block_Cipher_df**.

Process:

1. If (*number_of_bits_to_return* > *max_number_of_bits*), then return an **ERROR**.
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 bitsting 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.
7. K = Leftmost *keylen* bits of 0x010203...1F.
8. While **len** (*temp*) < *keylen* + *outlen*, do
   8.1 $IV = i \,||\, 0^{outlen\, -\, \mathbf{len}\, (i)}$. — Comment: The 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*.

9.6.4 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 guanteed by steps 4 and 8.1 in Section 9.6.3.

Output:

1. *output_block*: The result to be returned from the **Block_Cipher_Hash** operation.

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* ⊕ $block_i$.
   4.2 *chaining_value* = **Block_Encrypt** (*Key*, *input_block*).
5. *output_block* = chaining_value.
6. Return *output_block*.

9.7 Self-Testing of the DRBG

9.7.1 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.

Errors occurring during testing **shall** be perceived as complete DRBG failures. The condition causing the failure **shall** be corrected and the DRBG re-instantiated before requesting pseudorandom bits (also, see Section 9.8)

9.7.2 Testing the Instantiate Function

Whenever the instantiate function is invoked, known-answer tests on the instantiate function **shall** be performed prior to creating an operational instantiation. The *security_strength, prediction_resistance_flag* and *DRBG_specific_parameters* used in the 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** be 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 instantiate function may be used to instantiate using the tested values of *security_strength, prediction_resistance_flag* and *DRBG_specific_parameters.*

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

9.7.3 Testing the Generate Function

The generate function **shall** be tested upon power-up and at periodic intervals. The interval between periodic tests **shall** be consistent with the environment in which the DRBG is used. Note that in some environments, the periodic tests may need to be delayed until after a critical event has concluded; in this case, the periodic test **shall** be performed at the earliest possible opportunity.

Known-answer tests **shall** be performed on the generate function using 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.

9.7.4 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 (see Sections 8.2.3 and 10) **shall** be used. 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.

The reseed function may be called every time that the generate function is called if prediction resistance is available, and considerbly less frequently otherwise. In particular :

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.

9.7.5 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 "emptied". The reseed function **shall** be tested:

9.8 Error Handling

The expected errors are indicated for each DRBG function (see Sections 9.2 - 9.5) and for the derivation functions in Section 9.6. The error handling routines **should** indicate the type of error. For catastrophic errors (e.g., entropy input source failure), the DRBG **shall not** produce further output until the source of the error is corrected.

Many errors during normal operation may be caused by an application's improper DRBG request. In these cases, the 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.

Failures that indicate that the entropy source has failed or that the DRBG failed health testing (see Sections 9.7 and 11.4) **shall** be handled as complete DRBG failures. The indicated DRBG problem **shall** be corrected, and the DRBG **shall** be re-instantiated before the DRBG can be used to produce pseudorandom bits.

10 DRBG Algorithm Specifications

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 **shall** be conducted in order to select an appropriate DRBG. A detailed discussion on DRBG selection is provided in Annex E. Pseudocode examples for each DRBG are provided in Annex F. Conversion specifications required for the DRBG implementations (e.g., between integers and bitstrings) are provided in Annex B.

10.1 Deterministic RBGs Based on Hash Functions

10.1.1 Discussion

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

1. The **Hash_df_DRBG** specified in Section 10.1.2.
2. The **HMAC_DRBG** specified in Section 10.1.3.

The maximum security strength that could be supported by each hash function is provided in SP 800-57. However, this Standard supports only four security strengths: 112, 128, 192, and 256. Table 3 specifies the values that **shall** be used for the function envelopes and DRBG algorithm for each Approved hash function. The specifications in this Standard assume that a single appropriate hash function will be selected for a DRBG implementation; i.e., a DRBG implementation will not contain multiple hash functions from which to choose during instantiation.

Table 3: Definitions for Hash-Based DRBGs

	SHA-1	**SHA-224**	**SHA-256**	**SHA-384**	**SHA-512**
Supported security strengths	See SP 800-57				
highest_supported_security_strength	See SP 800-57				
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				

	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512
Seed length (*seedlen*) for Hash_df_DRBG	368	368	368	816	816
Maximum personalization string length (*max_personalization_string_length*)	$\leq 2^{35}$ bits				
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 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.

10.1.2 Hash_DRBG

10.1.2.1 Discussion

Figure 8 presents the normal operation of the **Hash_DRBG**. The **Hash_DRBG** requires the use of a hash function during the instantiate, reseed and generate functions; the same hash function **shall** be used in all functions. The hash function to be used **shall** meet or exceed the desired security strength of the consuming application.

Implementation validation testing and health testing are discussed in Sections 9.7 and 11.

10.1.2.2 Specifications

10.1.2.2.1 Hash_DRBG Internal State

The *internal_state* for **Hash_DRBG** consists of:

1. The *working_state*:
   a. A value (*V*) of *seedlen* bits that is updated during each call to the DRBG.
   b. A constant *C* of *seedle* bits that depends on the *seed*.
   c. A counter (*reseed_counter*) that indicates the number of requests for pseudorandom bits since new *entropy_input* was obtained during 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 *C* are the critical values of the internal state upon which the security of this DRBG depends (i.e., *V* and *C* are the "secret values" of the internal state).

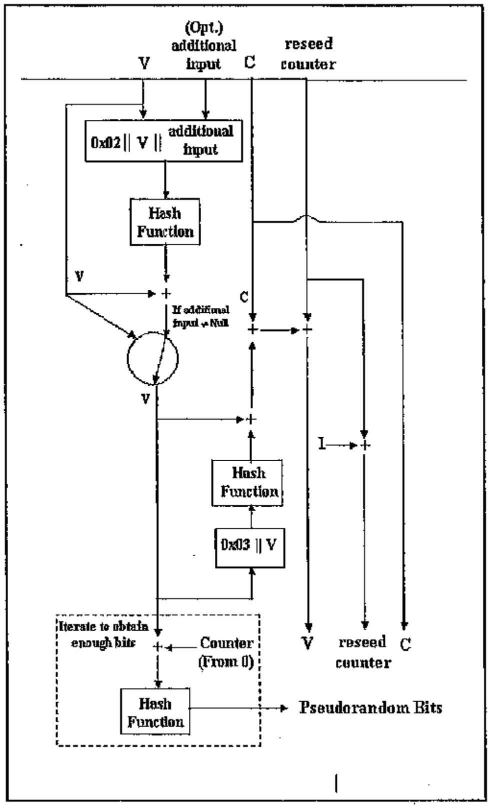

Figure 8: Hash_DRBG

10.1.2.2.2 Instantiation of Hash_DRBG

Notes for the instantiate function:

The instantiation of **Hash_DRBG** requires a call to the instantiate function specified in Section 9.2; step 9 of that function calls the instantiate algorithm in this section. For this DRBG, no *DRBG_specific_input_parameters* are required for the instantiate function specified in Section 9.2 (i.e., step 5 **should** be omitted).

The values of *highest_supported_security_strength* and *min_length* are provided in Table 3 of Section 10.1.1. The contents of the internal state are provided in Section 10.1.2.2.1.

The instantiate algorithm:

Let **Hash_df** be the hash derivation function specified in Section 9.6.2 using the selected hash function. The output block length (*outlen*), seed length (*seedlen*) and appropriate *security_strengths* for the implemented hash function are provided in Table 3 of Section 10.1.1.

The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG (see step 9 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.
3. *personalization_string*: The personalization string received from the consuming application. If a *personalization_string* will never be used, then steps 1 and 2 may be combined as follows:

   seed = **Hash_df** (*entropy_input*, *seedlen*).

Output:

1. *working_state*: The inital values for *V*, *C* and *reseed_counter* (see Section 10.1.2.2.1).

Process:

1. *seed_material* = *entropy_input* || *nonce* || *personalization_string*.
2. *seed* = **Hash_df** (*seed_material*, *seedlen*).
3. *V* = *seed*.
4. *C* = **Hash_df** ((0x00 || *V*), *seedlen*). Comment: Preceed *V* with a byte of zeroes.
5. *reseed_counter* = 1.
6. Return *V*, *C* and *reseed_counter* as the *working_state*.

10.1.2.2.3 Reseeding a Hash_DRBG Instantiation

Notes for the reseed function:

The reseeding of a **Hash_DRBG** instantiation requires a call to the reseed function specified in Section 9.3; 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 10.1.1.

The reseed algorithm:

Let **Hash_df** be the hash derivation function specified in Section 9.6.2 using the selected hash function. The value for *seedlen* is provided in Table 3 of Section 10.1.1.

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

Input:

1. *working_state*: The current values for *V*, *C* and *reseed_counter* (see Section 10.1.2.2.1).
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 provided, then step 2 may be

modified to remove the *additional_input*.

Output:

1. *status*: The status of the reseed function. The returned *status* is either **SUCCESS** or **ERROR**.
2. *working_state*: The new values for *V*, *C* and *reseed_counter*.

Process:

1. *V_old* = *V*.
2. *seed_material* = 0x01 || *V* || *entropy_input* || *additional_input*.
3. *seed* = **Hash_df** (*seed_material*, *seedlen*).
4. *V* = *seed*.
5. If (*V* = *V_old*), then return an **ERROR**.
6. *C* = **Hash_df** ((0x00 || *V*), *seedlen*). Comment: Preceed with a byte of all zeros.
7. *reseed_counter* = 1.
8. Return *V*, *C* and *reseed_counter* as the new *working_state*.

10.1.2.2.4 Generating Pseudorandom Bits Using Hash_DRBG

Notes for the generate function:

The generation of pseudorandom bits using a **Hash_DRBG** instantiation requires a call to the generate function specified in Section 9.4; 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 3 of Section 10.1.1.

The generate algorithm:

Let **Hash** be the selected hash function. The seed length (*seedlen*) and the maximum interval between reseeding (*reseed_interval*) are provided inTable 3 of Section 10.1.1. Note that for this DRBG, the reseed counter is used to update the value of *V* as well as to count the number of generation requests.

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

Input:

1. *working_state*: The current values for *V*, *C* and *reseed_counter* (see Section 10.1.2.2.1).
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 provided, then step 3 may be omitted.

Output:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS**, **ERROR**, 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. *working_state*: The new values for *V*, *C* and *reseed_counter*.

Process:

1. *V_old* = *V*.

2. If *reseed_counter* > *reseed_interval*, then return an indication that a reseed is required.

3. If (*additional_input* ≠ *Null*), then do

   3.1 w = **Hash** (0x02 || V || *additional_input*).

   3.2 $V = (V + w) \bmod 2^{seedlen}$.

4. *returned_bits* = **Hashgen** (*requested_number_of_bits*, *V*).

5. H = **Hash** (0x03 || V).

6. $V = (V + H + C + reseed\_counter) \bmod 2^{seedlen}$.

7. If (V = *V_old*), return an **ERROR**.

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

9. Return **SUCCESS**, *returned_bits*, and the new values of *V*, *C* and *reseed_counter* for the new *working_state*.

Hashgen (...):

Input:

1. *requested_no_of_bits*: The number of bits to be returned.

2. *V*: The current value of *V*.

Output:

1. *returned_bits*: The generated bits to be returned to the generate function.

Process:

1. $m = \left\lceil \frac{requested\_no\_of\_bits}{outlen} \right\rceil$.

2. *data* = *V*.

3. W = the *Null* string.

4. For i = 1 to m

   4.1 w_i = **Hash** (*data*).

   4.2 $W = W \| w_i$.

   4.3 $data = (data + 1) \bmod 2^{seedlen}$.

5. *returned_bits* = Leftmost (*requested_no_of_bits*) bits of W.

6. Return returned_bits.

10.1.3 HMAC_DRBG (...)

10.1.3.1 Discussion

HMAC_DRBG uses multiple occurrences of an Approved keyed hash function, which is based on an Approved hash function. The same hash function **shall** be used throughout. The hash function used **shall** meet or exceed the security requirements of the consuming application.

Figure 9 depicts the **HMAC_DRBG** in stages. **HMAC_DRBG** is specified using an internal function (**Update**). This function is called during the **HMAC_DRBG** instantiate, generate and reseed algorithms to adjust the internal state when new entropy or additional input is provided. The operations in the top portion of the figure are only performed if the additional input is not null. Figure 10 depicts the **Update** function.

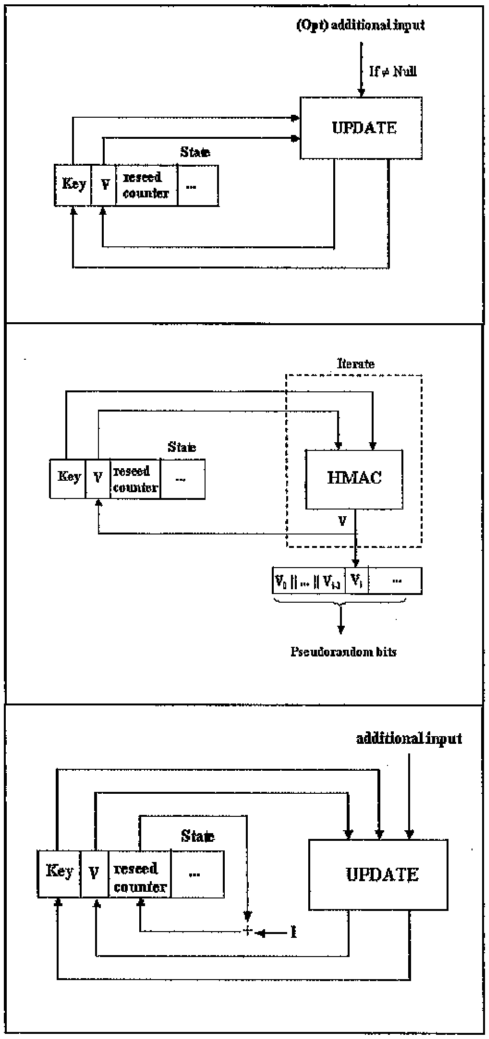

Figure 9: HMAC_DRBG

10.1.3.2 Specifications

10.1.3.2.1 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 3 of Section 10.1.1).
   b. The *Key* of *outlen* bits, 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.

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).

10.1.3.2.2 The Update Function (Update)

The **Update** function updates the internal state of **HMAC_DRBG** using the *provided_data*. Let **HMAC** be the keyed hash function specified in FIPS 198 using the hash function selected for the DRBG from Table 3 in Section 10.1.1.

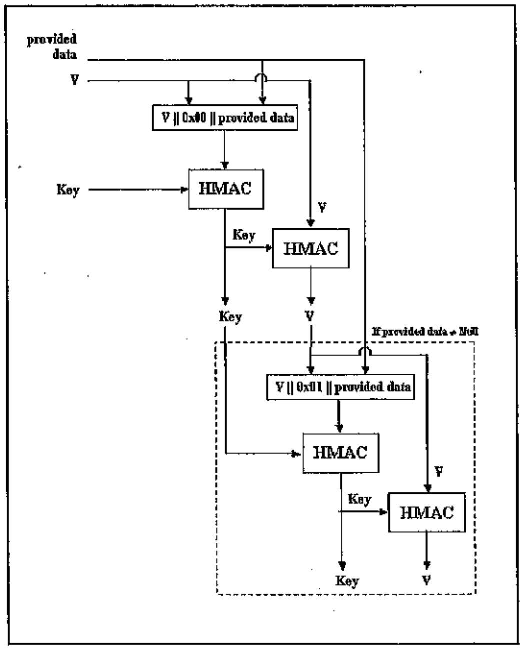

Figure 10: HMAC_DRBG Update Function

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*.

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.

10.1.3.2.3 Instantiation of HMAC_DRBG

Notes for the instantiate function:

The instantiation of **HMAC_DRBG** requires a call to the instantiate function specified in Section 9.2; step 9 of that function calls the instantiate algorithm specified in this section. For this DRBG, no *DRBG_specific_input_parameters* are required for the instantiate function specified in Section 9.2 (i.e., step 5 **should** be omitted). The values of *highest_supported_security_strength* and *min _length* are provided in Table 3 of Section 10.1.1. The contents of the internal state are provided in Section 10.1.2.2.1.

The instantiate algorithm:

Let **Update** be the function specified in Section 10.1.3.2.2. The ouput block length (*outlen*) is provided in Table 3 of Section 10.1.1.

The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG (see step 8 of 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.
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. *status*: The status returned from the instantiate function, where *status* is either **SUCCESS** or **ERROR**.
2. *working_state*: The inital values for V, *Key* and *reseed_counter* (see Section 10.1.3.2.1).

Process:

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

   Comment: Update *Key* and V.
4. (*Key*, V) = **Update** (*seed_material*, *Key_old*, *V_old*).

5. If ((*Key* = *Key_old*) or (*V* = *V_old*)), then return an **ERROR**.
6. *reseed_counter* = 1.
7. Return **SUCCESS**, *V*, *Key* and *reseed_counter* as the initial *working_state*.

10.1.3.2.4 Reseeding an HMAC_DRBG Instantiation

Notes for the reseed function:

The reseeding of an **HMAC_DRBG** instantiation requires a call to the reseed function specified in Section 9.3; 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 10.1.1.

The reseed algorithm:

Let **Update** be the function specified in Section 10.1.3.2.2. The following process or its equivalent **shall** be used as the reseed algorithmn for this DRBG (see step 5 of Section 9.3):

Input:

1. *working_state*: The current values for *V*, *Key* and *reseed_counter* (see Section 10.1.3.2.1).
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 step 1 may bemodified to remove the *additional_input*.

Output:

1. *status*: The status returned from the reseed function. The *status* is either **SUCCESS** or an **ERROR**.
2. *working_state*: The new values for *V*, *Key* and *reseed_counter*.

Process:

1. *V_old* = *V*; *Key_old* = *Key*.
2. *seed_material* = *entropy_input* || *additional_input*.
3. (*Key*, *V*) = **Update** (*seed_material*, *Key_old*, *V_old*).

   Comment: Check for "stuck"bits.
4. If ((*V* = *V_old*) or (*Key* = *Key_old*)), then return an **ERROR**.
5. *reseed_counter* = 1.
6. Return **SUCCESS**, *V*, *Key* and *reseed_counter* as the new *working_state*.

10.1.3.2.5 Generating Pseudorandom Bits Using HMAC_DRBG

Notes for the generate function:

The generation of pseudorandom bits using an **HMAC_DRBG** instantiation requires a call to the generate function specified in Section 9.4; 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 3 of Section 10.1.1.

The generate algorithm :

Let **HMAC** be the keyed hash function specified in FIPS 198 using the hash function selected for the DRBG. The value for *reseed_interval* is defined in Table 3 of Section 10.1.1.

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

Input:

1. *working_state*: The current values for *V_old*, *Key_old* and *reseed_counter* (see Section 10.1.3.2.1).
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 2 may be omitted. If *additional_input* is not provided (regardless of whether or not it will ever be provided), 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**, an **ERROR** 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. *working_state*: The new values for *V*, *Key* and *reseed_counter*.

Process:

1. If *reseed_counter* > *reseed_interval*, then return an indication that a reseed is required.
2. If *additional_input* ≠ *Null*, then (*Key_old*, *V_old*) = **Update** (*additional_input*, *Key_old*, *V_old*).
3. *temp* = *Null*.
4. While (**len** (*temp*) < *requested_number_of_bits*) do:

4.1 V = **HMAC** (*Key_old*, *V_old*).

Comment: Check for stuck bits.

4.2 If (V = *V_old*), then return an **ERROR**.

4.3 *V_old* = V.

4.4 *temp* = *temp* || V.

5. *returned_bits* = Leftmost *requested_number_of_bits* of *temp*.
6. (*Key*, V) = **Update** (*additional_input*, *Key_old*, *V_old*).

Comment: Check for "stuck" bits.

7. If ((V = *V_old*) or (*Key* = *Key_old*)), then return an **ERROR**.
8. *reseed_counter* = *reseed_counter* + 1.
9. Return **SUCCESS**, *returned_bits*, and the new values of *Key*, V and *reseed_counter* as the *working_state*).

10.2 DRBGs Based on Block Ciphers

10.2.1 Discussion

A block cipher DRBG is based on a block cipher algorithm. The block cipher DRBGs specified in this Standard have been designed to use any Approved block cipher algorithm and may be used by 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. The following are provided as DRBGs based on block cipher algorithms:

1. The **CTR_DRBG** specified in Section 10.2.2.
2. The **OFB_DRBG** specified in Section 10.2.3.

Table 4 specifies the values that **shall** be used for the function envelopes and DRBG algorithm for each Approved block cipher algorithm. The specifications in this Standard assume that a single appropriate block cipher algorithm and key size will be selected for a DRBG implementation; i.e., a DRBG implementation will not contain multiple block cipher algorithms or key sizes from which to choose during instantiation.

Table 4: Definitions for Block Cipher- Based DRBGs

	3 Key TDEA	AES-128	AES-192	AES-256
Supported security strengths	See SP 800-57			
highest_supported_security_strength	See SP 800-57			
Output block length (*outlen*)	64	128	128	128
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
A derivation function is used:				
Minimum entropy input length (*min _length*)	*security_strength*			
Maximum entropy input length (*max _length*)	$\leq 2^{35}$ bits			
Maximum personalization string length (*max_personalization_string_length*)	$\leq 2^{35}$ bits			
Maximum additional_input length (*max_additional_input_length*)	$\leq 2^{35}$ bits			

	3 Key TDEA	AES-128	AES-192	AES-256
A derivation function is not used (full entropy is available):				
Minimum entropy input length (*min _length*) (*outlen* + *keylen*)	*seedlen*			
Maximum entropy input length (*max _length*) (*outlen* + *keylen*)	*seedlen*			
Maximum personalization string length (*max_personalization_string_length*)	*seedlen*			
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 block cipher DRBGs may be implemented to use the block cipher derivation function specified in Section 9.6.3. However, these DRBGs are specified to allow an implementation tradeoff with respect to the use of this derivation function. If a source for full entropy input is always available to provide entropy input when requested, the use of the derivation function is optional; otherwise, the derivation functon **shall** be used. Table 4 provides lengths required for the *entropy_input*, *personalization_string* and *additional_input* for each case.

When full entropy is available, and a derivation function is not used by an implementation, the seed construction (seeSection 8.4.2) **shall not** use a nonce[3].

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.

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

10.2.2 CTR_DRBG

10.2.2.1 Discussion

CTR_DRBG uses an Approved block cipher algorithm in the counter mode as specified in [SP 800-38A]. 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. The values to be used for the implementation of this DRBG are specified in Table 4 of Section 10.2.1.

CTR_DRBG is specified using an internal function (**Update**). Figure 11 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. Figure 12 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.

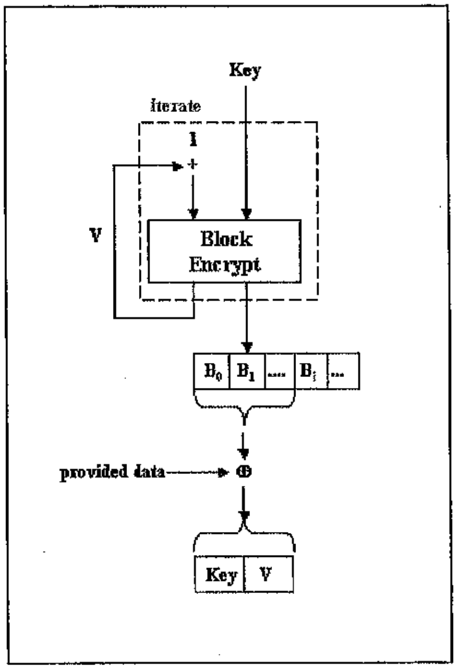

Figure 11: CTR_DRBG Update

10.2.2.2 Specifications

10.2.2.2.1 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 4 in Section 10.2.1).
   b. The *Key* of *keylen* bits, 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).

10.2.2.2.2 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 4 of Section 10.2.1. The block cipher operation in step 2.2 uses the selected block cipher algorithm.

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.

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*.

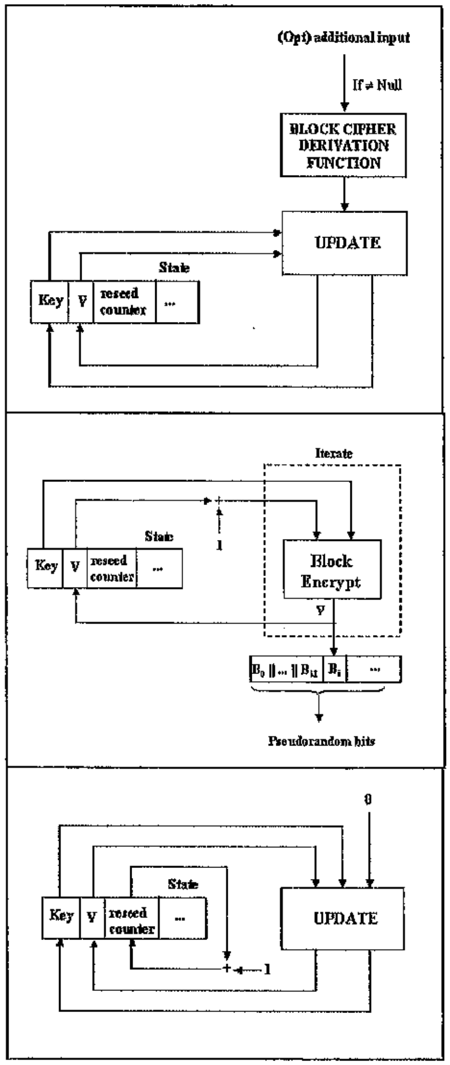

Figure 12: CTR_DRBG

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.

10.2.2.2.3 Instantiation of CTR_DRBG

Notes for the instantiate function:

The instantiation of **CTR_DRBG** requires a call to the instantiate function specified in Section 9.2; step 9 of that function calls the instantiate algorithm specified in this section. For this DRBG, no *DRBG_specific_input_parameters* are required for the instantiate function specified in Section 9.2 (i.e., step 5 **should** be omitted). The values of *highest_supported_security_strength* and *min _length* are provided in Table 4 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, and let **Block_Cipher_df** be the derivation function specified in Section 9.6.3 using the chosen block cipher algorithm and key size. The output block length (*outlen*), key length (*keylen*), seed length (*seedlen*) and *security_strengths* for the block cipher algorithms are provided in Table 4 of Section 10.2.1.

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

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; this string **shall not** be present when a derivation function is not used.
3. *personalization_string*: The personalization string received from the consuming application.

Output:

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

Process:

1. If the block cipher derivation function is available, then

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

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

Else — Comment: The block cipher derivation function is not used and full entropy is known to be available.

1.3 *temp* = **len** (*personalization_string*).

1.4 If *temp* > *seedlen*, then return an **ERROR**.

1.5 If (*temp* < *seedlen*), then *personalization_string* = *personalization_string* || $0^{seedlen\ -\ temp}$.

1.6 *seed_material* = *entropy_input* ⊕ *personalization_string*.

2. *Key* = 0^{keylen}. — Comment: *keylen* bits of zeros.
3. *V* = 0^{outlen}. — Comment: *outlen* bits of zeros.
4. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).
5. *reseed_counter* = 1.
6. Return *V*, *Key* and *reseed_counter* as the *working_state*.

Implementation notes:

1. Step 1 should consist of either steps 1.1 and 1.2, or steps 1.3 – 1.6. The decision for the substeps to be used depends on whether the implementation has full entropy and is using the derivation function.
2. If a *personalization_string* will never be provided from the instantiate function and a derivation function will be used, then step 1.1 becomes:

   seed_material = **Block_Cipher_df** (*entropy_input*, *seedlen*).
3. If a *personalization_string* will never be provided from the instantiate function, a full entropy source will be available and a derivation function will not be used, then step 1 becomes

   seed_material = *entropy_input*.

   That is, steps 1.3 – 1.6 collapse into the above step.

10.2.2.2.4 Reseeding a CTR_DRBG Instantiation

Notes for the reseed function:

The reseeding of a **CTR_DRBG** instantiation requires a call to the reseed function specified in Section 9.3; step 5 of that function calls the reseed algorithm specified in this section. The values for *min _length* are provided in Table 4 of Section 10.2.1.

The reseed algorithm:

Let **Update** be the function specified in Section 10.2.2.2.2, and let **Block_Cipher_df** be the derivation function specified in Section 9.6.3 using the chosen block cipher algorithm and key size. The seed length (*seedlen*) is provided in Table 4 of Section 10.2.1.

The following process or its equivalent **shall** be used as the reseed algorithm for this DRBG (see step 5 of 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.

Output:

1. *status*: The status returned from the instantiate function. The *status* is either **SUCCESS** or an **ERROR**.
2. *working_state*: The new values for *V*, *Key* and *reseed_counter*.

Process:

1. If the block cipher derivation function is available, then
   1.1 *seed_material* = *entropy_input* || *additional_input*.
   1.2 *seed_material* = **Block_Cipher_df** (*seed_material*, *seedlen*).

   Else — Comment: The block cipher derivation function is not used because full entropy is known to be available.

   1.3 *temp* = **len** (*additional_input*).
   1.4 If *temp* > *seedlen*, then return an **ERROR**.
   1.5 If (*temp* < *seedlen*), then *additional_input* = *additional_input* || $0^{seedlen\ -\ temp}$.
   1.6 *seed_material* = *entropy_input* ⊕ *additional_input*.
2. *V_old* = *V*; *Key_old* = *Key*.
3. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).
4. If ((*V* = *V_old*) or (*Key* = *Key_old*)), then return an **ERROR**.
5. *reseed_counter* = 1.
6. Return *V*, *Key* and *reseed_counter* as the *working_state*.

Implementation notes:

1. Step 1 should consist of either steps 1.1 and 1.2, or steps 1.3 – 1.6. The decision for the substeps to be used depends on whether the implementation has full entropy and is using the derivation function.
2. If *additional_input* will never be provided from the reseed function and a derivation function will be used, then step 1.1 becomes:

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

3. If *additional_input* will never be provided from the reseed function, a full entropy source will be available and a derivation function will not be used, then step 1 becomes

   seed_material = *entropy_input*.

   That is, steps 1.3 – 1.6 collapse into the above step.

10.2.2.2.5 Generating Pseudorandom Bits Using CTR_DRBG

Notes for the generate function:

The generation of pseudorandom bits using a **CTR_DRBG** instantiation requires a call to the generate function specified in Section 9.4, 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 4 of Section 10.2.1. If the derivation function is not used, then the maximum allowed length of *additional_input* = *seedlen*.

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

Let **Block_Cipher_df** be the derivation function specified in Section 9.6.3, and let **Update** be the function specified in Section 10.2.2.2.2 using the chosen block cipher algorithm and key size. The seed length (*seedlen*) and the value of *reseed_interval* are provided in Table 4 of Section 10.2.1. Step 4.2 below uses the selected block cipher algorithm. If a derivation function is not used for a DRBG implementation, then step 2.2 **shall** be omitted.

The following process or its equivalent **shall** be used as generate algorithm for this DRBG (see step 8 of 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 *additional_input* will never be provided, then step 3 may be omitted.

Output:

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

Process:

1. *V_old* = *V*. *Key_old* = *Key*.
2. If *reseed_counter* > *reseed_interval*, then return an indication that a reseed is required.
3. If (*additional_input* ≠ *Null*), then

   Comment: If the length of the *additional input* is > *seedlen*, derive *seedlen* bits.

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

   Comment: If a block cipher derivation function is used:

   3.2 If (*temp* > *seedlen*), then *additional_input* = **Block_Cipher_df** (*additional_input*, *seedlen*).

   Comment: If the length of the *additional_input* is < *seedlen,* pad with zeros to *seedlen* bits.

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

   3.4 (*Key*, *V*) = **Update** (*additional_input*, *Key*, *V*).
4. *temp* = *Null*.
5. While (**len** (*temp*) < *requested_number_of_bits*) do:

   5.1 $V = (V + 1) \bmod 2^{outlen}$.

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

   5.3 *temp* = *temp* || *ouput_block*.
6. *returned_bits* = Leftmost *requested_number_of_bits* of *temp*.

Comment: Update for backtracking

resistance.

7. $zeros = 0^{seedlen}$. Comment: Produce a string of *seedlen* zeros.
8. (*Key*, *V*) = **Update** (*zeros*, *Key*, *V*).
9. If ((*V* = *V_old*) or (*Key* = *Key_old*)), then return an **ERROR**.
10. *reseed_counter* = *reseed_counter* + 1.
11. Return **SUCCESS** and *returned_bits*; also return *Key*, *V* and *reseed_counter* as the new *working_state*.

10.2.3 OFB_DRBG

10.2.3.1 Discussion

OFB_DRBG uses an Approved block cipher algorithm in the output feedback mode as specified in [SP 800-38A]. 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. The values to be used for the implementation of this DRBG are specified in Table 4 in Section 10.2.1.

OFB_DRBG is specified using an internal function (**Update**). Figure 13 depicts the **OFB_DRBG** in three stages. The operations in the top portion of the figure are only performed if non-null additional input is provided. Figure 14 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.Note that **OFB_DRBG** is basically the same as **CTR_DRBG,** except that the block cipher mode is OFB rather than CTR.

10.2.3.2 Specifications

10.2.3.2.1 OFB_DRBG Internal State

The internal state for **OFB_DRBG** consists of:

1. The working_state:
   a. The value *V*, which is updated each time another *outlen* bits of output are produced.
   b. The *Key*, which is updated whenever a predetermined number of output blocks are generated.
   c. A counter (*reseed_counter*) that

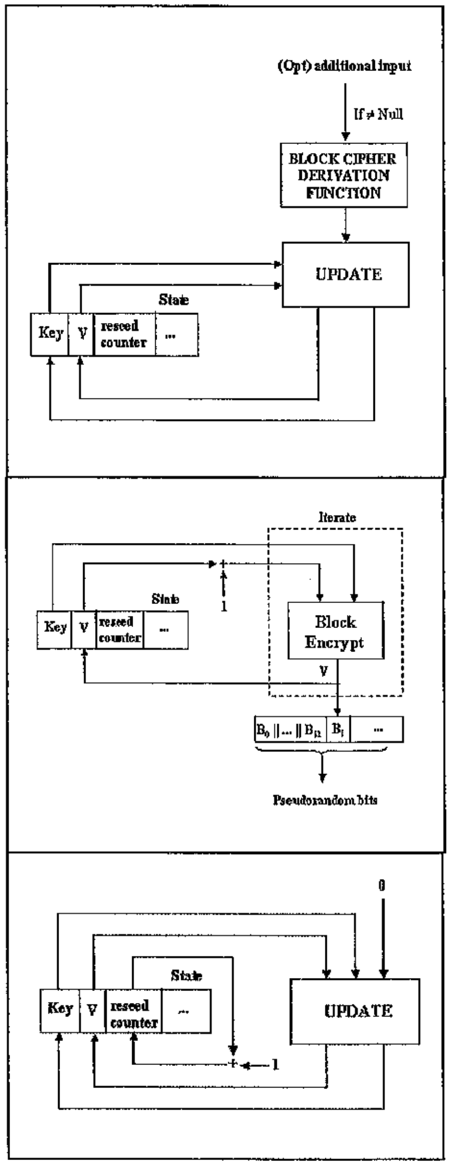

Figure 13: OFB_DRBG

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).

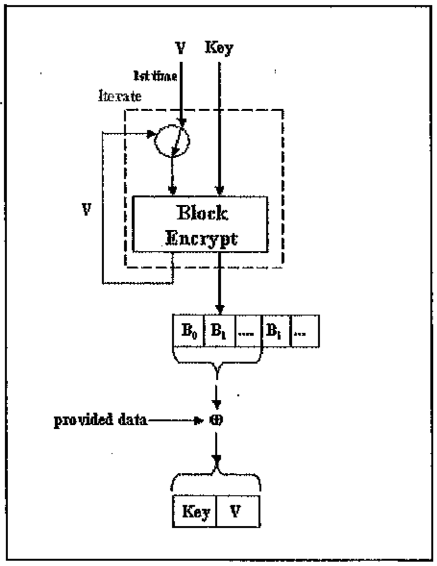

Figure 14: OFB_DRBG Update

10.2.3.2.2 The Update Function(Update)

The **Update** function updates the internal state of the **OFB_DRBG** using the *provided_data.* The values for *outlen, keylen* and *seedlen* are provided in Table 4 of Section 10.2.1. The block cipher operation in step 2.1 uses the selected block cipher algorithm and key size.

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

Input:

1. *provided_data*: The data to be used.
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*.

Process:

1. *temp* = *Null.*
2. While (**len** (*temp*) < *seedlen*) do
   2.1 *V* = **Block_Encrypt** (*Key*, *V*).
   2.2 *temp* = *temp* || *V*.

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.

10.2.3.2.3 Instantiation of OFB_DRBG (...)

This process is the same as the instantiation process for **CTR_DRBG** in Section 10.2.2.2.3, except that the **Update** function to be used is specified in Section 10.2.3.2.2.

10.2.3.2.4 Reseeding an OFB_DRBG Instantiation

This process is the same as the reseeding process for **CTR_DRBG** in Section 10.2.2.2.4, except that the **Update** function to be used is specified in Section 10.2.3.2.2

10.2.3.2.5 Generating Pseudorandom Bits Using OFB_DRBG

This process is the same as the generation process for **CTR_DRBG** in Section 10.2.2.2.5, except that the **Update** function to be used is specified in Section 10.2.3.2.2 and step 5 **shall** be as follows:

5. While (**len** (*temp*) < *requested_number_of_bit*) do:

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

    52 *temp* = *temp* || *V*.

10.3 Deterministic RBGs Based on Number Theoretic Problems

10.3.1 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 10.3.2 specifies a DRBG based on the elliptic curve discrete logarithm problem; Section 10.3.3 specifies a DRBG based on a problem related to the RSA problem of finding roots modulo a composite integer.

10.3.2 Dual Elliptic Curve Deterministic RBG (Dual_EC_DRBG)

10.3.2.1 Discussion

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 the NIST curves given in this Standard, $m \geq 163$. Figure 15 depicts the **Dual_EC_DRBG**.

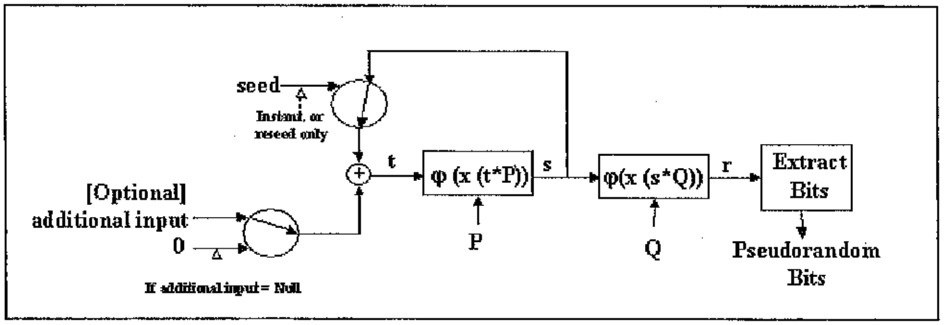

Figure 15: Dual_EC_DRBG

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* + 64 bits. Further requirements for the *seed* are provided in Section 8.4.

Backtracking resistance is inherent in the algorithm, even if the internal state is compromised. As shown in Figure 16, **Dual_EC_DRBG** generates a *seedlen*-bit number for each step $i = 1,2,3,\ldots$, as follows:

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

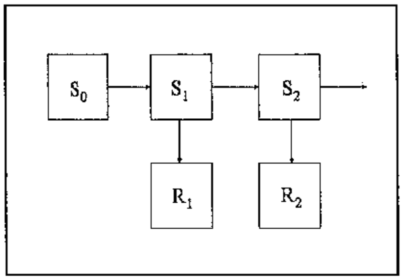

Figure 16: Dual_EC_DRBG (...) Backtracking Resistance

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, and by truncation to produce the output. 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 5 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 except the first three 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 5: Definitions for the Dual_EC_DRBG

	P-224	B-233	K-233	P-256	B-283	K-283
Supported security strengths	See SP 800-57					
highest_supported_ security_strength	See SP 800-57					
Output block length (*outlen* = largest multiple of 8 less than *seedlen* - (13 + $\log_2$ (the cofactor))	208	216	216	240	264	264
Required minimum entropy for instantiate and reseed	*security_strength*					
Minimum entropy input length (*min_length* = 8 × ⌈*seedlen*/8⌉)	224	240	240	256	288	288
Maximum entropy input length (*max_length*)	$\leq 2^{13}$ bits					

Comment [ebb6]: Page: 78
Why can't this be min_entropy ?

	P-224	B-233	K-233	P-256	B-283	K-283
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* = *m*)	224	233	233	256	283	283
Appropriate hash functions	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512					
max_number_of_bits_per_request	*outlen* × *reseed_interval*					
Number of blocks between reseeding (*reseed_interval*)	≤ 10,000 blocks					

	P-384	B-409	K-409	P-521	B-571	K-571
Supported security strengths	See 800-57					
highest_supported_ security_strength	See SP 800-57					
Output block length (*outlen* = smallest multiple of 8 less than *seedlen* - (13 + $\log_2$ (the cofactor))	368	392	392	504	552	552
Required minimum entropy for instantiate and reseed	*security_stength*					
Minimum entropy input length (*min _length* = $8 \times \lceil seedlen/8 \rceil$)	384	416	416	528	576	576
Maximum entropy input length (*max _length*)	$\leq 2^{13}$ bits					
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* = *m*)	384	409	409	521	571	571
Appropriate hash functions	SHA-224, SHA-256, SHA-384, SHA-512			SHA-256, SHA-384, SHA-512		
max_number_of_bits_per_request	*outlen* × *reseed_interval*					
Number of blocks between reseeding (*reseed_interval*)	≤ 10,000 blocks					

Validation and Operational testing are discussed in Section 11. Detected errors **shall** result in a transition to the error state.

10.3.2.2 Specifications

10.3.2.2.1 Dual_EC_DRBG Internal State and Other Specification Details

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 (*curve_type*, *seedlen*, p, a, b, n), where *curve_type* indicates a prime field F_p, or a pseudorandom or Koblitz curve over the binary field F_{2^m}; *seedlen* is the length of the seed ; 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*. If only one type of curve is implemented, the *curve_type* parameter may be omitted.
   c. Two points P and Q on the curve; the generating point G specified in FIPS 186-3 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, and

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).

10.3.2.2.2 Instantiation of Dual_EC_DRBG

Notes for the instantiate function:

The instantiation of **Dual_EC_DRBG** requires a call to the instantiate function specified in Section 9.2; step 9 of that function calls the instantiate algorithm in this section. For this DRBG, a DRBG-specific input parameter of *requested_curve_type* is optional (see the definition for *curve_type* in Section 10.3.2.2.1). If only one type of curve is available, then this parameter may be omitted. If multiple types are available, then a *Prime_field_curve* will be selected if the parameter is omitted; if a

Prime_field_curve is not available, then a *Random_binary_curve* will be selected.

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

5. Using *requested_curve_type* (if provided), the *security_strength* and Table 5 in Section 10.3.2.1, select the smallest available curve that has a security strength ≥ *security_strength*.

   5.1 If *requested_curve_type* is indicated, then select a curve of that type. If no suitable curve of that type is available for the *requested_security_strength*, then return an **ERROR**.

   5.2 If a curve type is not requested, then select an appropriate *Prime_field_curve* if a suitable curve is available. If no suitable *Prime_field curve* is available, then select a *Random_binary_curve* if a suitable curve is available. If no suitable *Random_binary_curve* is available, then select a *Koblitz_curve*. If no suitable *Koblitz_curve* is available, then return an ERROR.

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

The values for *highest_supported_security_strength* and *min_length* are determined by the selected curve (see Table 5 in Section 10.3.2.1).

The instantiate algorithm :

Let **Hash_df** be the hash derivation function specified in Section 9.6.2 using an appropriate hash function from Table 5 in Section 10.3.2.1. Let *seedlen* be the appropriate value from Table 5.

The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG (see step 9 of 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.
3. *personalization_string*: The personalization string received from the consuming application.

Output:

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

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*).

Comment: Save all state information.

3. *block_counter* = 0.
4. Return *s* and *block_counter* for the *working_state*.

Implementation notes:

If an implementation never uses a *personalization_string*, then steps 1 and 2 may be combined as follows :

s = **Hash_df** (*entropy_input*, *seedlen*).

10.3.2.2.3 Reseeding of a Dual_EC_DRBG Instantiation

Comment [ebb7]: Page: 82
Need to add steps to perform the « continuous » test.

Notes for the reseed function:

The reseed of **Dual_EC_DRBG** requires a call to the reseed function specified in Section 9.3; step 5 of that function calls the reseed algorithm in this section. The values for *min _length* are provided in Table 5 of Section 10.3.2.1.

The reseed algorithm :

Let **Hash_df** be the hash derivation function specified in Section 9.6.2 using an appropriate hash function from Table 5 in Section 10.3.2.1.

The following process or its equivalent **shall** be used to reseed the **Dual_EC_DRBG** process after it has been instantiated (see step 5 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. *status*: The status returned from the reseed function. The *status* is either **SUCCESS** or **ERROR**.
2. *s*: The new value of the secret parameter in the *working_state*.
3. *block_counter*: The re-initialized block counter for reseeding.

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_string*.
2. *s_old* = *s*.
3. *s* = **Hash_df** (*seed_material*, *seedlen*).
4. If (*s* = *s_old*), then return an **ERROR**.
5. *block_counter* = 0.
6. 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*.

10.3.2.2.4 Generating Pseudorandom Bits Using Dual_EC_DRBG

Notes for the generate function:

The generation of pseudorandom bits using a **Dual_EC_DRBG** instantiation requires a call to the generate function specified in Section 9.4; 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 4 of Section 10.2.1.

The generate algorithm:

Let **Hash_df** be the hash derivation function specified in Section 9.6.2 using an appropriate hash function from Table 5 in Section 10.3.2.1. The value of *reseed_interval* is also provided in Table 5.

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.

d. φ (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. Section 10.3.2.2.4 has the details of this mapping.

The precise definition of $\varphi(x)$ used in steps 6 and 7 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 m = *seedlen*):

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

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

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

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

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

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

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 in Section 9.4):

Input:

1. *working_state*: The current values for *s, curve_type, seedlen, p, a, b, n, P, Q* and *reseed_counter* (see Section 10.1.3.2.1).
2. *requested_number_of_bits*: The number of pseudorandom bits to be returned to the generate functione.
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, ERROR** 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 *working_state*.
4. *block_counter*: The updated block counter for reseeding.

Process:

Comment: Check whether a reseed is required.

1. If $\left(block\_counter + \left\lceil \dfrac{requested\_number\_of\_bits}{outlen} \right\rceil \right) > reseed\_interval$, 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. $t = s \oplus additional\_input$.
6. $s\_old = s$.
7. $s = \varphi(x(t * P))$.

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

8. If ($s = s\_old$), then return an **ERROR**.
9. $r = \varphi(x(s * Q))$. Comment: r is a *seedlen*-bit number.
10. *temp* = *temp* || (**rightmost** *outlen* bits of r).
11. *additional_input*=0

    Comment: *seedlen* zeroes; *additional_input_string* is added only on the first iteration.

12. *block_counter* = *block_counter* + 1.
13. $i = i + 1$.
14. If (**len** (*temp*) < *requested_number_of_bits*), then go to step 6.
15. *returned_bits* = **Truncate** (*temp*, $i \times outlen$, *requested_number_of_bits*).
16. Return **SUCCESS**, *returned_bits*, and *s* and *block_counter* for the *working_state*.

10.3.3 Micali-Schnorr Deterministic RBG (MS_DRBG)

10.3.3.1 Discussion

The **MS_DRBG** generalizes the RSA generator, which is defined as follows: Let **gcd**(x, y) denote the greatest common divisor of the integers x and y, and $\phi(n)$ represent the Euler phi function[4]. Select n, the product of two distinct large primes, and e, a positive integer such that **gcd**(e, $\phi(n)$) = 1. Define $f(y) = y^e \; mod \; n$. Starting with a seed y_0, form the sequence $y_{i+1} = f(y_i)$, and output the string consisting of the lg lg (n) least significant bits of each y_i. These bits are known to be as secure as the RSA function f, and are commonly referred to as the *hard* bits.

The Micali-Schnorr generator **MS_DRBG** uses the same e and n as the RSA generator, but produces many more random bits per iteration and eliminates the overlap between the state sequence and the output bits. Each $y_i \in [0, n)$ is viewed as the concatenation $s_i \,\|\, z_i$ of an r-bit number s_i and a $k = \lg(n)\text{-}r$ bit number z_i. The s_i are used to propagate the integer sequence $y_{i+1} = s_i^e \; mod \; n$; the z_i are output as random bits. r must be at least 2*min{*security_strength*, lg(n)/e}, where *security_strength* is the desired security strength of the generator, and $e \geq 65{,}537$. (See Section 10.3.3.2.2). A random r-bit *seed* s_0 is used to initialize the process.

Figure 17 depicts the **MS_DRBG**. Under the proper assumption, the **MS_DRBG** is an example of a cryptographically secure generator, i.e., one that passes all polynomial-time statistical tests. The assumption is that sequences of the form $s^e \; mod \; n$ are statistically the same as sequences of integers in Z_n. This assumption is stronger than requiring the intractability of the RSA problem. See [1] for a discussion of these concepts and references to further details.

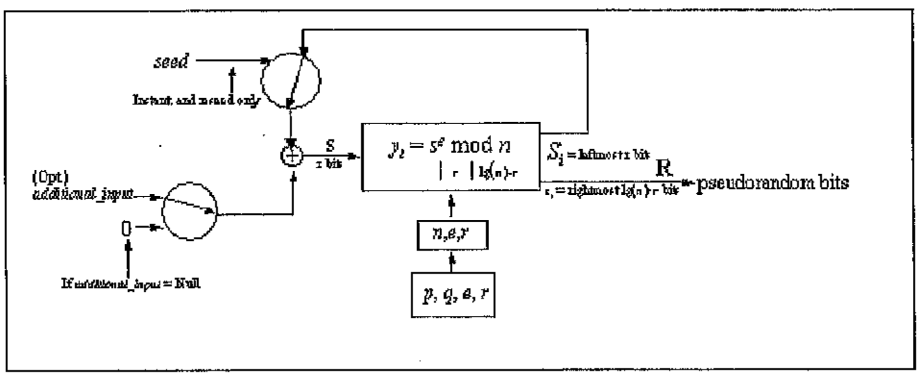

Figure 17: MS_DRBG

[4] The Euler phi function : $\phi(n)$ = the number of positive integers < n that are relatively prime to n. For an RSA modulus $n = pq$, $\phi(n) = (p\text{-}1)(q\text{-}1)$.

For **MS_DRBG**, the *s* values are assumed to be *r*-bit integers, and "statistically the same" means indistinguishable by any polynomial-time algorithm. Accepting the stronger assumption allows *k* to be a significant percentage of lg(*n*). Note that in the specifications, *r* has been redefined as *seedlen*, and *k* has been redefined to be *outlen* in order to be consistent with the other DRBGs.

The specifications for the **MS_DRBG** (see Section 10.3.3.2) allow *e* and *k* (i.e., *outlen*) to be specified. The lengths *seedlen* and *outlen*, the RSA modulus *n*, and the value of the exponent *e* are variable within the bounds described below. The bounds are based on the desired *security strength* for the bits produced. For maximum efficiency, *e* **should** be kept small and *outlen* **should** be large. The *outlen* bits generated at each step are concatenated to form pseudorandom bitstrings of any desired length. Table 6 provides definitions for using with the **MS_DRBG** functions and algorithms.

Table 6: Definitions for MS_DRBG

	lg (*n*) = 2048	**lg (n) = 3072**
Supported security strengths	See SP 800-57	
highest_supported_security_strength	See SP 800-57	
Output Block Length (*outlen* = *k*)	$8 \le outlen \le$ **min**{ lg(*n*) – 2**security_strength*, lg(*n*) – 2*lg(*n*)/*e*	
Required minimum entropy for instantiate and reseed	*Security_strength*	
Minimum entropy input length (*min _length*)	*security_strength*	
Maximum entropy input length (*max _length*)	$\le 2^{13}$ bits	
Maximum personalization string length (*max_personalization_string_length*)	$\le 2^{13}$ bits	
Maximum additional_input length (*max_additional_input_length*)	$\le 2^{13}$ bits	
Number of hard bits (lg (lg (*n*))	11	11
Seed length (*seedlen* = *r*)	lg(*n*) – *outlen*	
Appropriate hash functions	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512	
max_number_of_bits_per_request	*outlen* × *reseed_interval*	
Number of blocks of *outlen* between reseeds (*reseed_interval*)	≤ 50,000 blocks	

10.3.3.2 MS_DRBG Specifications

10.3.3.2.1 Internal State for MS_DRBG

The internal state for MS_DRBG consists of:

1. The *working_state*:
   a. The M-S parameters n, e, *seedlen* and *outlen*, and
   b. An integer S in $[0,2^{seedlen})$ that propagates the internal state sequence from which pseudorandom bits are derived.
   c. A counter (*block_counter*) that indicates the number of blocks of random produced by **MS_DRBG** during the current instance since the previous reseeding.
2. Administrative information:
   a. The *security_strength* provided by the instance of the DRBG, and
   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).

10.3.3.2.2 Selection of the M-S parameters

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

The M-S parameters n, *seedlen*, e and *outlen* are selected to satisfy the following six conditions, based on *strength*:

1. $1 < e < \phi(n)$; $\mathbf{gcd}(e, \phi(n)) = 1$. Comment: ensures that the mapping $s \rightarrow s^e \bmod n$ is 1-1.
2. $(e \times seedlen) \geq 2*\lg(n)$. Comment: ensures that the exponentiation requires a full modular reduction.
3. $seedlen \geq 2*security\_strength$. Comment: protects against a tableization attack.
4. *outlen* and *seedlen* are multiples of 8. Comment: This is an implementation convenience.
5. $outlen \geq 8$; $seedlen + outlen = \lg(n)$. Comment: all bits are used.
6. $n = p*q$. Comment: p and q are strong [as in FIPS 186-

3], secret primes .

The M-S parameters are determined in this order:

1. The size of the modulus lg(n) is set first. It **shall** conform to the values given in Table 6 for the requested *security_strength*.

2. The RSA exponent e. The implementation **should** allow the application to request any odd integer e in the range $1 < e < 2^{\lg(n)-1} - 2*2^{½\lg(n)}$. [Comment: The inequality ensures that $e < \phi(n)$ when an Approved algorithm is used to generate the primes p and q.] If e is not provided during an instantiate request, or *requested_e* = 0 is supplied, the default value e=3 **should** be used.

Comment [ebb8]: Page: 89
For DSS, 16,537 < e < ($2^{nlen-2s}$-1), where nlen is the length of n, and s is the security strength.

3. The number *outlen* of output bits used for each iteration. The implementation **should** allow any multiple of 8 in the range 8 ≤ *outlen* ≤ min{ lg(n) – 2**security_strength*, lg(n) – 2*lg(n)/e } to be requested. However, if a value for *outlen* is not provided or *requested_outlen* = 0 is specified, *outlen* **should** be selected as the *largest* multiple of 8 integer in the allowable range **and** within the range of bits currently known to be *hard* bits for the RSA problem. That value is lg(lg(n)), as shown in Table 6. Thus, in all cases, the default value 8 will be used if *requested_outlen* = 0.

   Any values for *requested_e* and *requested_outlen* outside these ranges **shall** be flagged as **errors**.

4. Set the size of the seeds: *seedlen* = lg(n) – *outlen*.

5. Selection of the modulus n. Two primes p and q of size ½lg(n) bits, having entropy at least *min_entropy*, and satisfying **gcd** (e, (p-1)(q-1)) = 1 **shall** be generated as specified in FIPS 186-3. An implementation **shall** use strong primes as defined in that document: each of p-1, p+1, q-1, q+1 **shall** have a large prime factor of at least *security_strength* bits. [Comment: Any Approved algorithm will generate a modulus of size lg(n) bits using strong primes of size ½ lg(n) bits, and will allow the exponent e to be specified beforehand.]

The difficulty of the RSA problem relies on the secrecy of the primes p and q comprising the modulus. Whenever private primes are generated, the implementation **shall** clear memory of those values immediately after n has been computed. Only the modulus n **shall** be kept in the internal *state*.

10.3.3.2.3 Instantiation of MS_DRBG

Notes for the instantiate function:

The instantiation of **MS_DRBG** requires a call to the instantiate function specified in Section 9.2; step 8 of that function calls the instantiate algorithm in this section. For this DRBG, two DRBG-specific input parameters may be provided: *requested_e* and *requested_outlen.*

The values for *highest_supported_security_strength* and *min_length* are provided in

Table 6 in Section 10.3.3.1.

In step 5 of the instantiate function, the following steps **shall** be used to select values for *n, e, seedlen* and *outlen*:

5. Using *security_strength*, *requested_e* (if provided) and *requested_outlen* (if provided), select values for *n*, *e*, *seedlen* and *outlen*.

   Comment: Determine the modulus size.

   5.1 If *security_strength* = 112, then lg (n) = 2048

   Else lg (n) = 3072.

   Comment: Select the exponent e.

   5.2 If *requested_e* < 65537 or is not provided, then e = 65,537

   Else

   5.2.1 e = *requested_e*.

   5.2.2 If (*requested_e* < 3) or(*requested_e* > $2^{\lg(n)-1} - (2 \times 2^{1/2 \lg(n)})$) or (*requested_e* is even), then return an **ERROR**.

   Comment : Select the output length *outlen.*

   5.3 If *requested_outlen* = 0 or is not provided, then *outlen* = 8

   Else

   5.3.1 *outlen* = *requested_outlen.*

   5.3.2 If (*outlen* < 1) or (outlen > **min** ($\lfloor \lg (n) - 2 \times security\_strength \rfloor$, $\lfloor \lg (n) \times (1 - 2/e) \rfloor$) or (outlen is not a multiple of 8), then return an **ERROR**.

   Comment : Determine the seed length (*seedlen*).

   5.4 *seedlen* = lg (n) - *outlen*.

   Comment: Get the modulus n.

   5.5 Using lg (n) and e, get a random modulus n. n **shall** be the product of two primes p and q such that :

   1) Each has a length of lg (n)/2 bits,

   2) Each has at least *security_strength* + 64 bits of entropy,

   3) **gcd** (e, (p-1), (q-1)) = 1.

   4) (p-1), (p+1), (q-1) and (q+1) **shall** each have a large prime factor of at least *security_strength* bits.

5.6 $n = p \times q$.

5.7 $p = q = 0$.

Since the values for *working_state* values *n*, *e*, and *outlen* have been determined by step 5 (above), they need not be provided to nor returned from the instantiate algorithm in step 9; however, the value of *seedlen* is required by the instantite algorithm and must be provided to it.

The instantiate algorithm:

Let **Hash (...)** be an Approved hash function for the security strengths to be supported.

The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG (see step 9 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.
3. *personalization_string*: The personalization string received from the consuming application.
4. *seedlen*: The length of the seed.

Output:

1. *working_state*: The inital values for *S* and *block_counter* (see Section 10.3.3.2.1).

Process:

1. *seed_material* = *entropy_input* || *nonce* || *personalization_string*.
2. *S* = **Hash_df** (*seed_material*, *seedlen*).
3. *block_counter* = 0.
4. Return **SUCCESS**, *S* and *block_counter* for the *working_state*.

Implementation notes:

If a *personalization_string* will never be provided, then steps 1 and 2 may be combined as follows:

S = **Hash_df** (*entropy_input*, *seedlen*).

10.3.3.2.4 Reseeding of a MS_DRBG Instantiation

Notes for the reseed function:

The reseed of **MS_DRBG** requires a call to the reseed function specified in Section 9.3; step 5 of that function calls the reseed algorithm in this section. The values for *min_length* are provided in Table 6 of Section 10.3.3.1.

The reseed algorithm:

Let **Hash_df** be the hash derivation function specified in Section 9.6.2 using an appropriate hash function from Table 6 in Section 10.3.3.1.

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

Input:

1. *working_state*: The current values for *seedlen* and *S*.
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. *status*: The status of performing this algorihm. The *status* is either **SUCCESS** or **ERROR.**
2. *working_state*: The new values for *S* and *block_counter*.

Process:

1. *seed_material* = *S* || *entropy_input* || *additional_input*.
2. *S_old* = *S*.
3. *S* = **Hash_df** (*seed_material*, *seedlen*).
4. If (*S* = *S_old*), then return an **ERROR.**
5. *block_counter* = 0.
6. Return **SUCCESS**, and the new values of *S* and *block_counter*.

Implementation notes:

If *additional_input* will never be provided, then steps 1 may be modified as follows:

seed_material = *S* || *entropy_input*.

10.3.3.2.5 Generating Pseudorandom Bits Using MS_DRBG

Notes for the generate function:

The generation of pseudorandom bits using an **MS_DRBG** instantiation requires a call to the generate function specified in Section 9.4; 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 6 of Section 10.3.3.1.

The generate algorithm:

Let **Hash_df** be the hash derivation function specified in Section 9.6.2 using an appropriate hash function from Table 6 in Section 10.3.3.1. The value of *reseed_interval* is also specified in Table 6.

Let **pad8** (*bitstring*) be a function that inputs an arbitrary length *bitstring* and returns a copy of that *bitstring* padded on the right with binary 0's, if necessary, to a multiple of 8. Note: This is an implementation convenience for byte-oriented functions.

Let **Truncate** (*bits, in_len, out_len*) be a function that inputs a bitstring of *in_len* bits, returning a string consisting of the leftmost *out_len* bits of input. If *in_len* < *out_len*, the input string is returned padded on the right with *out_len* – *in_len* zeroes.

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

Input:

1. *working_state*: The current values for *n, e, seedlen, outlen, S,* and *reseed_counter* (see Section 10.3.3.2.1).
2. *requested_number_of_bits*: The number of pseudorandom bits to be returned to the generate functione.
3. *additional_input*: The additional input string received from the consuming application.

Output:

1. *status*: The status returned from thefunction. The *status* will indicate **SUCCESS,** an **ERROR** 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 updated secret value in the *working_state*.
4. *block_counter*: The updated block counter for reseeding.

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.

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

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

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

Comment: Hash to *seedlen* bits.

Comment: Produce *requested_number_of_bits*, *outlen* at a time.

3. *temp* = the Null string.
4. $i = 0$.
5. $S\_old = S$.
6. $s = S \oplus additional\_input$. Comment: s is to be interpreted as a *seedlen*-bit unsigned integer.
7. $S = \lfloor (s^e \bmod n) / 2^{outlen} \rfloor$ Comment: S is a *seedlen*-bit number.
8. If ($S = S\_old$), then return **ERROR**.
9. $R = (s^e \bmod n) \bmod 2^{outlen}$. Comment: R is an *outlen*-bit number.
10. $temp = temp \,||\, R$.
11. $additional\_input = 0^{seedlen}$. Comment: *seedlen* zeroes.
12. $i = i + 1$.
13. $block\_counter = block\_counter + 1$.
14. If (**len** (*temp*) < *requested_number_of_bits*), then go to step 6.
15. *returned_bits* = **Truncate** (*temp*, $i \times k$, *requested_number_of_bits*).
16. Return **SUCCESS**, *returned_bits* and the values of S and *block_counter* for the *working_state*.

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 18.

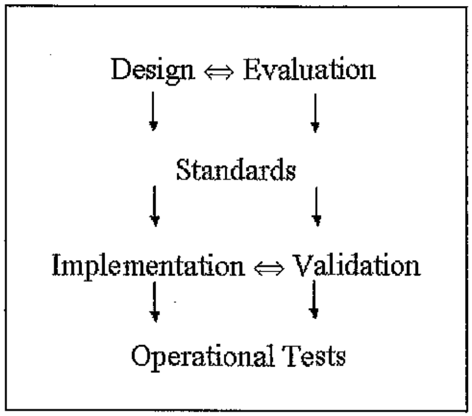

Figure 18: DRBG Assurance Strategy

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, but this Standard requires the development of basic documentation to provide minimal assurance that the DRBG process has been implemented properly (see Section 11.2). An implementation **should** be validated for conformance to this Standard by an accredited laboratory (see Section 11.3). 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.

Operational (i.e., 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.

A cryptographic module containing a DRBG **should** be validated (see FIPS 140-2 [8]). 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.

Note that any entropy input used for testing (either for validation testing or operational/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:

Comment [ebb9]: Page: 96
Probably need to add additional documentation requirements to address other requirements.

- Document how the implementation has been designed to permit implementation validation and operational testing.
- Document the type of DRBG (e.g., Hash_DRBG, Dual_EC_DRBG), and the cryptographic primitives used (e.g., SHA-256, AES-128).
- 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 and OFB_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 when full entropy input is available.
- Document any support functions other than operational 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 Operational/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 prior to the first instantiation and periodically, and a capability to perform self-tests on demand **shall** be included (see Section 9.7). 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 Sections 7.2.7 and 9.8).

11.4.2 Known Answer Testing

Known answer testing **shall** be conducted prior to the first instantiation and periodically, and may be conducted on demand. 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 Sections 7.2.7 and 9.8).

The generalized known answer testing is specified in Section 9.7. 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 NIST approved curves and points **shall** be used in applications requiring certification under FIPS 140-2. More details about these curves may be found in FIPS PUB 186-3, the Digital Signature Standard.

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 (...)**

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-224

```
p = 26959946667150639794667015087019630673557916\
    260026308143510066298881

r = 26959946667150639794667015087019625940457807\
    714424391721682722368061

b = b4050a85 0c04b3ab f5413256 5044b0b7 d7bfd8ba 270b3943
    2355ffb4


Px = b70e0cbd 6bb4bf7f 321390b9 4a03c1d3 56c21122 343280d6
     115c1d21

Py = bd376388 b5f723fb 4c22dfe6 cd4375a0 5a074764 44d58199
     85007e34
```

Qx = 68623591 6e11adfa f080a451 477fa27a f21248be 916d3458
a583a3c9

Qy = 6060018a 24b35be6 caecf3f0 7f2c6b43 4e47479e 55362c8f
5707adca

A.1.1.2 Curve P-256

p = 1157920892103562487626974469494075735300861\
3415290314195533631308867097853951

r = 11579208921035624876269744694940757352999695\
5224135760342422259061068512044369

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.1.1.3 Curve P-384

p = 39402006196394479212279040100143613805079739\
270465444666794829340424572177149687032904726\
6088258938001861606973112319

r = 39402006196394479212279040100143613805079739\
270465444666794690527962765939911326356939895\
6308152294913554433653942643

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

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

Py = 3617de4a 96262c6f 5d9e98bf 9292dc29 f8f41dbd 289a147c

```
e9da3113 b5f0b8c0 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.1.1.4 Curve P-521

p = 68647976601306097149819007990813932172694353\
001433054093944634591855431833976560521225596\
406614545549772963113914808580371219879997166\
4381257402829111505715 1

r = 68647976601306097149819007990813932172694353\
001433054093944634591855431833976553942450577\
463332171975329639963713633211138647686124403\
80340372808892707005449

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.1.2 Curves over Binary Fields

For each field degree m, a pseudo-random curve (B) and a Koblitz curve (K) are given.

The pseudo-random curve has the form

E: $y^2 + xy = x^3 + x^2 + b$,

and the Koblitz curve has the form

E: $y^2 + xy = x^3 + ax^2 + 1$, where $a = 0$ or 1.

For each pseudorandom curve, the cofactor is $f = 2$. The cofactor of each Koblitz curve is $f = 2$ if $a = 1$, and $f = 4$ if $a = 0$.

The coefficients of the pseudo-random curves, and the coordinates of the points P and Q for both kinds of curves, are given in terms of both the polynomial and normal basis representations, in hex.

NOTE: An implementation may choose to represent coordinates in either basis. However, in order to gain certification it must demonstrate agreement with the test output vectors, which have been generated using the normal basis representation for each of the binary curves.

The order r of the base point P is given 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()**. r is given in decimal

A.1.2.1 Curve K-233

```
a = 0

r = 34508731733952818937173779311385127605709409888622521\
    26328087024741343
```

Polynomial Basis:

```
Px = 00000172 32ba853a 7e731af1 29f22ff4 149563a4 19c26bf5
     0a4c9d6e efad6126

Py = 000001db 537dece8 19b7f70f 555a67c4 27a8cd9b f18aeb9b
     56e0c110 56fae6a3
```

Normal Basis:

```
Px = 000000fd e76d9dcd 26e643ac 26f1aa90 1aa12978 4b71fc07
     22b2d056 14d650b3

Py = 00000064 3e317633 155c9e04 47ba8020 a3c43177 450ee036
     d6335014 34cac978
```

Polynomial Basis:

```
Qx = 000000aa 7178e973 8a6f797a 1c265465 06106896 0a58b3fe
     a3afc77f 18404eee

Qy = 0000002d 12a8f3e9 884bf31d 052a8eaf 414b891a 0a40491e
     1f9d2576 79248ee2
```

Normal Basis:

Qx = 0000015a 96493d91 e56b5f10 579a7d58 eb895e06 8d94e1af
86d34143 4377548c

Qy = 0000006b 13a689bb 3730dfd7 a46486ea ff8eb6cb 9d815981
a927d2eb 8cfa9b00

A.1.2.3 Curve B-233

r = 69017463467905637874347558622770255558398127373450135\
55379383634485463

Polynomial Basis:

b = 066 647ede6c 332c7f8c
0923bb58 213b333b 20e9ce42 81fe115f 7d8f90ad

Px = 000000fa c9dfcbac 8313bb21 39f1bb75 5fef65bc 391f8b36
f8f8eb73 71fd558b

Py = 00000100 6a08a419 03350678 e58528be bf8a0bef f867a7ca
36716f7e 01f81052

Normal Basis:

b = 1a0 03e0962d 4f9a8e40
7c904a95 38163adb 82521260 0c7752ad 52233279

Px = 0000018b 863524b3 cdfefb94 f2784e0b 116faac5 4404bc91
62a363ba b84a14c5

Py = 00000049 25df77bd 8b8ff1a5 ff519417 822bfedf 2bbd7526
44292c98 c7af6e02

Polynomial Basis:

Qx = 000000cb 50ce04af f4ea6111 aaccfe04 ae5f0dfe 95a59db4
cd4aba0c 1126615a

Qy = 0000005b ab8a93a0 5c42caae 1b322b14 876ec2e0 5c994a25
8e67295e 5808eaf9

Normal Basis:

Qx = 00000055 ea07c1ca 4a4312f3 4562737c 257f4fa8 3b9d3d48
8a123cab 238f69a2

Qy = 00000055 d60ea17a 1cb969a8 3786a82f 8172e889 026195f9

923ba4b1 beeb5702

A.1.2.2 Curve K-283

a = 0

r = 38853377844514581418389238136470378132848117337930613\
2429587499752981582970442260387 3

Polynomial Basis:

Px = 0503213f 78ca4488 3f1a3b81 62f188e5 53cd265f 23c1567a
16876913 b0c2ac24 58492836

Py = 01ccda38 0f1c9e31 8d90f95d 07e5426f e87e45c0 e8184698
e4596236 4e341161 77dd2259

Normal Basis:

Px = 03ab9593 f8db09fc 188f1d7c 4ac9fcc3 e57fcd3b db15024b
212c7022 9de5fcd9 2eb0ea60

Py = 02118c47 55e7345c d8f603ef 93b98b10 6fe8854f feb9a3b3
04634cc8 3a0e759f 0c2686b1

Polynomial Basis:

Qx = 0388eee4 1cc5808d 140d5179 76fba0fa 9c14b886 914387a6
890a9497 fd3370b6 9cdd3779

Qy = 04d86b99 fed2ecad 1dc9fd77 ed5928ac ef908f97 1eb22cf6
8e436df4 dbe6e06e b2c2dff4

Normal Basis:

Qx = 004ab17d 72374eb7 dac733d8 83d7b650 eb03ccb9 d6c60197
74f41ef2 1b8e0e11 0fe8aa58

Qy = 07243a25 e2e7e633 7897e8b1 9791c813 0317aecf 8c0ac2a4
2ac03dac 4afdabe8 ffc9888c

A.1.2.4 Curve B-283

r = 77706755689029162836778476272940756265696259243769048\
89109196526770044277787378692871

Polynomial Basis:

b = 27b680a c8b8596d a5a4af8a 19a0303f
ca97fd76 45309fa2 a581485a f6263e31 3b79a2f5

Px = 05f93925 8db7dd90 e1934f8c 70b0dfec 2eed25b8 557eac9c

80e2e198 f8cdbecd 86b12053

Py = 03676854 fe24141c b98fe6d4 b20d02b4 516ff702 350eddb0
826779c8 13f0df45 be8112f4

Normal Basis:

b = 157261b 894739fb 5a13503f 55f0b3f1
0c560116 66331022 01138cc1 80c0206b dafbc951

Px = 0749468e 464ee468 634b21f7 f61cb700 701817e6 bc36a236
4cb8906e 940948ea a463c35d

Py = 062968bd 3b489ac5 c9b859da 68475c31 5bafcdc4 ccd0dc90
5b70f624 46f49c05 2f49c08c

Polynomial Basis:

Qx = 06530328 33283d9e b6ebc03c 2d735ed9 12b46bc1 2e364643
f8e309d9 d55e9440 28190ba5

Qy = 03693cd3 8b4e022d ef81bb7f 949ca7f4 287cbc3d 3aae8632
a6fea719 e0da9998 48211443

Normal Basis:

Qx = 06c2366c 8acc000a 5b516dfc 4cf8a204 b255dd0d e53f18e1
99718e05 47b3845f 000626c9

Qy = 03667f53 e1e528e9 99bfb2cb 9e609116 969d78fb 94a264a9
a2045878 132ca8f5 85b874ef

A.1.2.5 Curve K-409

a = 0

r = 330527984395124299475957654016385519914202341482140609\
642324395022880711289249191050673258457777458014096366\
590617731358671

Polynomial Basis:

Px = 0060f05f 658f49c1 ad3ab189 0f718421 0efd0987 e307c84c
27accfb8 f9f67cc2 c460189e b5aaaa62 ee222eb1 b35540cf
e9023746

Py = 01e36905 0b7c4e42 acba1dac bf04299c 3460782f 918ea427
e6325165 e9ea10e3 da5f6c42 e9c55215 aa9ca27a 5863ec48
d8e0286b

Normal Basis:

```
Px = 01b559c7 cba2422e 3affe133 43e808b5 5e012d72 6ca0b7e6
     a63aeafb c1e3a98e 10ca0fcf 98350c3b 7f89a975 4a8e1dc0
     713cec4a

Py = 016d8c42 052f07e7 713e7490 eff318ba 1abd6fef 8a5433c8
     94b24f5c 817aeb79 852496fb ee803a47 bc8a2038 78ebf1c4
     99afd7d6
```

Polynomial Basis:

```
Qx = 01ba9a6c 2d31edf6 671ce7d1 f16f4ab2 7c72ca88 cc3b33e9
     b2ef536e 92bc06ad 0cac0d6a 821898c2 847b5d7e 8506fd26
     9e51dfcc

Qy = 019d9567 d1931672 ab748567 c4fb75a4 e0658b9b bf17901e
     b7d41148 489ab481 354977ac 390bbb05 a6e782b5 13caa159
     02a846ef
```

Normal Basis:

```
Qx = 00e8b595 6a3f2ec5 e8e3e3cf e4c2003a 687feecc ade301e5
     c34d47ef a723dac6 36f1ef6a cd5ced42 309fc937 fa5460d5
     223c3743

Qy = 001f61f2 2a66d942 de111925 dd94da7d 5c02e4c2 23328be5
     9019a157 d7b700f6 d8b42316 efe8193d 68c90ce0 fe57ad2b
     4f690281
```

A.1.2.6 Curve B-409

```
r = 661055968790248598951915308032771039828404682964281219\
    284648798304157774827374805208143723762179110965979867\
    288366567526771
```

Polynomial Basis:

```
b =  021a5c2 c8ee9feb 5c4b9a75
    3b7b476b 7fd6422e f1f3dd67 4761fa99 d6ac27c8
    a9a197b2 72822f6c d57a55aa 4f50ae31 7b13545f

Px = 015d4860 d088ddb3 496b0c60 64756260 441cde4a f1771d4d
     b01ffe5b 34e59703 dc255a86 8a118051 5603aeab 60794e54
     bb7996a7

Py = 0061b1cf ab6be5f3 2bbfa783 24ed106a 7636b9c5 a7bd198d
     0158aa4f 5488d08f 38514f1f df4b4f40 d2181b36 81c364ba
     0273c706
```

Normal Basis:

b = 124d065 1c3d3772 f7f5a1fe
6e715559 e2129bdf a04d52f7 b6ac7c53 2cf0ed06
f610072d 88ad2fdc c50c6fde 72843670 f8b3742a

Px = 00ceacbc 9f475767 d8e69f3b 5dfab398 13685262 bcacf22b
84c7b6dd 981899e7 318c96f0 761f77c6 02c016ce d7c548de
830d708f

Py = 0199d64b a8f089c6 db0e0b61 e80bb959 34afd0ca f2e8be76
d1c5e9af fc7476df 49142691 ad303902 88aa09bc c59c1573
aa3c009a

Polynomial Basis:

Qx = 01920ed2 5ec895fc 704ac0da 05a93ace 25fc9646 ab4533c0
4f759ce1 ac0e53d8 096b2318 d6fdd0d7 1d2affd6 915e8d7a
e2977127

Qy = 011d1d15 0c127a29 77b48a17 fac8aa13 96985213 3179fc17
74f9d3db 1f6bee43 d8c04cce 35f2abf8 022230f6 457f260a
72444bfd

Normal Basis:

Qx = 01b2481e 3265c48d 28db6172 95efafd5 77f7d0ed 175cc49b
0fcb1982 639bc380 eee80285 e6ef8a7b 1a31566d 602c07dc
dc85a5a5

Qy = 00d0712d 082d31ba 22497958 b1178993 a2f5dc41 f14207e4
0f8ccda8 06b637cc f1380320 b6ff9dfd 8e811f14 49c4c23e
2f4823fe

A.1.2.7 Curve K-571

a = 0

r = 1932268761508629172347675945465993672149463664853217\4\
99328617625725759571144780212268133978522706711834706\
71280082535146127367497406661731192968242161709250355\
5733685276673

Polynomial Basis:

Px = 026eb7a8 59923fbc 82189631 f8103fe4 ac9ca297 0012d5d4
60248048 01841ca4 43709584 93b205e6 47da304d b4ceb08c
bbd1ba39 494776fb 988b4717 4dca88c7 e2945283 a01c8972

```
Py = 0349dc80 7f4fbf37 4f4aeade 3bca9531 4dd58cec 9f307a54
     ffc61efc 006d8a2c 9d4979c0 ac44aea7 4fbebbb9 f772aedc
     b620b01a 7ba7af1b 320430c8 591984f6 01cd4c14 3ef1c7a3
```

Normal Basis:

```
Px = 004bb2db a418d0db 107adae0 03427e5d 7cc139ac b465e593
     4f0bea2a b2f3622b c29b3d5b 9aa7a1fd fd5d8be6 6057c100
     8e71e484 bcd98f22 bf847642 37673674 29ef2ec5 bc3ebcf7

Py = 044cbb57 de20788d 2c952d7b 56cf39bd 3e89b189 84bd124e
     751ceff4 369dd8da c6a59e6e 745df44d 8220ce22 aa2c852c
     fcbbef49 ebaa98bd 2483e331 80e04286 feaa2530 50caff60
```

Polynomial Basis:

```
Qx = 06c62ea8 63120582 6a8e4328 412a3400 0be7c23f 19982e7f
     35164b12 c18df503 2997173d 9776bab1 2dafe58e 97e1aa9d
     4726eaae 6473c2bc 7e0c2752 fed22ac2 e86fbcfc 00468dc4

Qy = 070b1c34 39bb9845 42f21349 21ff78d0 ce6efb9b f27f02b5
     0f83c658 f29b2076 ac77c8ac 015be59c 02d090fb 20aa4a35
     f4745614 78445d04 fd2ee388 3cbd5508 f7edcfe7 a803dd47
```

Normal Basis:

```
Qx = 01e8cee5 3c73b384 ad828269 7566e3ad b11573fd 7aff7abd
     1af60123 062e560c 1bb66d35 d00cd77e 101e7606 6afcd0c9
     8c8826eb 79b91e33 1328701c 9fb5c3ab 01d798af c4fbea67

Qy = 079d03ff 6f51d98d 4679aa59 97b51eca e2ecf2fe ba491edf
     d5df7df7 277bb265 b58b11ad 5b916e99 fea7ef78 49314df1
     0af703bd 1b202c8c fa97760b 27044c19 ac5d9fb5 65381df3
```

A.1.2.8 Curve B-571

```
r = 38645375230172583446953518909319873442989273297064349\
    98657235251451519142289560424536143999389415773083133\
    88112192694448624687246281681307023452828830333241139\
    3191105285703
```

Polynomial Basis:

```
b = 2f40e7e 2221f295 de297117
    b7f3d62f 5c6a97ff cb8ceff1 cd6ba8ce 4a9a18ad
    84ffabbd 8efa5933 2be7ad67 56a66e29 4afd185a
    78ff12aa 520e4de7 39baca0c 7ffeff7f 2955727a
```

```
Px = 0303001d 34b85629 6c16c0d4 0d3cd775 0a93d1d2 955fa80a
     a5f40fc8 db7b2abd bde53950 f4c0d293 cdd711a3 5b67fb14
     99ae6003 8614f139 4abfa3b4 c850d927 e1e7769c 8eec2d19

Py = 037bf273 42da639b 6dccfffe b73d69d7 8c6c27a6 009cbbca 1980f853
     3921e8a6 84423e43 bab08a57 6291af8f 461bb2a8 b3531d2f
     0485c19b 16e2f151 6e23dd3c 1a4827af 1b8ac15b
```

Normal Basis:

```
b =  3762d0d 47116006 179da356

    88eeaccf 591a5cde a7500011 8d9608c5 9132d434

    26101a1d fb377411 5f586623 f75f0000 1ce61198

    3c1275fa 31f5bc9f 4be1a0f4 67f01ca8 85c74777


Px = 00735e03 5def5925 cc33173e b2a8ce77 67522b46 6d278b65
     0a291612 7dfea9d2 d361089f 0a7a0247 a184e1c7 0d417866
     e0fe0feb 0ff8f2f3 f9176418 f97d117e 624e2015 df1662a8

Py = 004a3642 0572616c df7e606f ccadaecf c3b76dab 0eb1248d
     d03fbdfc 9cd3242c 4726be57 9855e812 de7ec5c5 00b4576a
     24628048 b6a72d88 0062eed0 dd34b109 6d3acbb6 b01a4a97
```

Polynomial Basis:

```
Qx = 01e263e6 afad323f 934e50e4 da0b015b 3f6727f4 27701cc3
     0dcd1145 c12e3c66 50ccd260 5ccd5a6a 609c5acd 3aed9e2d
     32de8e64 80303414 dc0907f0 21f8cefd cfb45700 56f8d686

Qy = 06c99cbb 0c686a6e d6b7015d e2cbe18a 3f623ae2 c87ab4a3
     d6cd7b78 b37f49cc 5e88de04 b5668dad 2df3f34c 50b8c56a
     3140d87f 81abb42e 919b3f8d 61743ba9 14bcb11b defda5cf
```

Normal Basis:

```
Qx = 01ece446 40b698fe eb575fc0 65156c5f f94c277a 5335e1a2
     28b65c22 aff27777 d159cfee c7f1270c c84bca33 8f34ab4d
     6748f592 bf322442 e2ffeffe 9e5a321d cd6b4e75 a269e745

Qy = 01cadda7 5647bba5 8c08b5e2 2b633e3a 5dd3b2c9 5db81f2d
     220cba3d 7a38e692 072b3db2 6465b27a 2abd56b4 2291f982
     3a902eb5 038d162a 7a578d37 8dd0c620 4f722521 b8084d4c
```

A.2 Test Moduli for the MS_DRBG (...)

Each modulus is of the form $n = pq$ with $p = 2p_1 + 1$, $q = 2q_1 + 1$, where p_1 and q_1 are $(\lg(n)/2 - 1)$-bit primes.

A.2.1 The Test Modulus *n* of Size 2048 Bits

The hexadecimal value of the modulus *n* is:

```
c11a01f2 5daf396a a927157b af6f504f 78cba324 57b58c6b
f7d851af 42385cc7 905b06f4 1f6d47ab 1b3a2c12 17d14d15
070c9da5 24734ada 2fe17a95 e600ae9a 4f8b1a66 96661e40
7d3043ec d1023126 5d8ea0d1 81cf23c6 dd3dec9e b3fce204
5b9299bb cca63dee 435a2251 ad0765d4 9d29db2e f5aba161
279aeb5f 6899fe48 7973e36c 1fb13086 d9231b6b 925a8495
4ba0fbca fea844ea 77a9f852 f86915a4 e71bd0ba b9b269c3
9a7a827a 41311ffa 4470140c 8b6509fe 5dbd39e3 ec816066
2d036e13 0e07e233 06a39b18 db0e8efe 64418880 81ac3673
2b4091f6 63690d03 3b486d74 371a20fc 3e214bce 7ed0e797
5ea44453 cd161d32 e8185204 59896571
```

A.2.2 The Test Modulus *n* of Size 3072 Bits

The hexadecimal value of the modulus *n* is:

```
c6046ba6 8beaa061 c468a9a7 4da34d64 21398c73 020837c7
d2a4042b dd9a7628 cab8022e 5bc4246f 75da8d26 03da8021
41c5d112 835e6bdb 57ed799e 28d6fa49 c3d0f5b5 f9776c14
0a901bf7 73ae3113 35d0470e da91b442 dbac621a cdd324e2
a70244d7 cb155adc 4b77dd94 fafe069d 5b5cc494 86e9fe61
c5081190 abb24f54 2d7d21e9 c90453c6 9ac63143 401d6b35
e456ea2f 64ae76f9 2df80328 b48f7962 d5c9b779 b2078496
7d374f02 06b8afbf 678d7f5f 36c3d84e c9e55c28 7ce5c668
17ee05b4 1059168f b5c5e2a3 6bc2f6ce 3b43bd14 56eebdd5
70ffe61e 5a7023a9 04d98f8a 96bfaf55 55a12f81 5561b401
63f3a50e a1e16a36 3f5cddd4 a1db275c 4fc2d650 d51f1e93
f5fd7631 ca45914f f6fe62a0 be55b997 5f6566bb 47e76276
f4e3b2eb 837bf0da 9d824687 042479a3 04147399 2d814a3a
7be7bc3d 06992df6 6c1d7d06 f8c1410e 2bbb573a 0e278e7a
daa600f3 2577030e 95b73dd9 96b65f98 4740a485 e27138bd
d5f02522 09bcf005 6640a1b3 b1dd97ad 7c187e04 01ba817d
```

ANNEX B : (Normative) Conversion and Auxilliary Routines

B.1 Bitstring to an Integer

Input:

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

Output:

1. x The requested integer representation of the bitstring.

Process:

1. Let $(b_1, b_2, ..., b_n)$ 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 to be converted.

Output:

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

Process:

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

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

[The information in this annex needs nto be reconsidered. Is C.1 needed here ? The information in C.2 is provided in SP 800-57. C.3 is needed only if Dual_EC_DRBG is retianed. What other information is appropriate ?]

C.1 The Security of Hash Functions

[Add a discussion as to why it is OK to use SHA-1 to generate pseudorandom curves of greater than 80 bits of security. The security strength of a hash function for these generators is = the output block size. If there is no vulnerability to collision (e.g., when a hash function is used as an element in a well-designed RNG) and the function is not invertible, than the strength is = the ouput block size. However, when a hash function is used as an element in an application/cryptographic service where vulnerability to collisions is a consideration, then the strength = half the size of the output block.]]

C.2 Algorithm and Keysize Selection

This section provides guidance for the selection of appropriate algorithms and key sizes. It emphasizes the importance of acquiring cryptographic systems with appropriate algorithms and key sizes to provide adequate protection for 1) the expected lifetime of the system and 2) any data protected by that system during the expected lifetime of the data. Also included is the necessity for selecting appropriate random bit generators to support the cryptographic algorithms.

Cryptographic algorithms provide different levels (i.e., different "strengths") of security, depending on the algorithm and the key size used. Two algorithms are considered to be of equivalent strength for the given key sizes (X and Y) if the amount of work needed to "break the algorithms" or determine the keys (with the given key sizes) is approximately the same using a given resource. The strength of an algorithm (sometimes called the work factor) for a given key size is traditionally described in terms of the amount of work it takes to try all keys for a symmetric algorithm with a key size of "X" that has no short cut attacks (i.e., the most efficient attack is to try all possible keys). In this case, the best attack is said to be the exhaustion attack. An algorithm that has a "Y" bit key, but whose strength is equivalent to an "X" bit key of such a symmetric algorithm is said to provide "X bits of security" or to provide "X-bits of strength". An algorithm that provides X bits of strength would, on average, take $2^{X-1}T$ to attack, where T is the amount of time that is required to perform one encryption of a plaintext value and comparison of the result against the corresponding ciphertext value.

Determining the security strength of an algorithm can be nontrivial. For example, consider TDEA. TDEA uses three 56-bit keys ($K1$, $K2$ and $K3$). If each of these keys is independently generated, then this is called the three key option or three key TDEA (3TDEA). However, if $K1$ and $K2$ are independently generated, and $K3$ is set equal to $K1$, then this is called the two key option or two key TDEA (2TDEA). One might expect that 3TDEA would provide $56 \times 3 = 168$ bits of strength. However, there is an attack on 3TDEA that reduces the strength to the work that would be involved in exhausting a 112-bit key. For 2TDEA, if exhaustion were the best attack, then the strength of 2TDEA would be $56 \times 2 = 112$ bits. This appears to be the case if the attacker has only a few matched plain and cipher pairs. However, if the attacker can obtain approximately 2^{40} such pairs, then 2TDEA has strength that is comparable to an 80-bit algorithm (see [ASCX9.52], Annex B) and, therefore, is not appropriate for this Standard, since the lowest security strength provides 112 bits of security.

The comparable key sizes discussed in this section are based on assessments made as of the publication of this Standard. Advances in factoring algorithms, advances in general discrete logarithm attacks, elliptic curve discrete logarithm attacks and quantum computing may affect these assessments in the future. New or improved attacks or technologies may be developed that leave some of the current algorithms completely insecure. If quantum computing becomes a practical reality, the asymmetric techniques may no longer be secure. Periodic reviews will be performed to determine whether the stated comparable sizes need to be revised (e.g., the key sizes need to be increased) or the algorithms are no longer secure.

When selecting a block cipher cryptographic algorithm (e.g., AES or TDEA), the block size may also be a factor that should be considered, since the amount of security provided by several of the modes defined in [SP 800-38] is dependent on the block size[5]. More information on this issue is provided in [SP 800-38].

Table 7 provides associated key sizes for the Approved algorithms and hash functions.

1. Column 1 indicates the security strength provided by the algorithms and key sizes in a particular row.
2. Column 2 provides the symmetric key algorithms that provide the indicated level of security (at a minimum), where TDEA is approved in [ASC X9.52], and AES is specified in [FIPS 197]. The table entry for TDEA requires the use of three distinct keys.
3. Column 3 provides the comparable security strengths for hash functions that are specified in FIPS180-2. The hash function entries assume that collision resistance is required (e.g., the application uses the hash function for digital signatures). For applications that are not concerned with collisions, the appropriate application standard will specify the appropriate hash functions for the security level. For this Standard, see Section 10.1.1 and Table 3.

[5] Suppose that the block size is b bits. The collision resistance of a MAC is limited by the size of the tag and collisions become probable after $2^{b/2}$ messages, if the full b bits are used as a tag. When using the Output Feedback mode of encryption, the maximum cycle length of the cipher can be at most 2^b blocks; the average cipher length is less than 2^b blocks. When using the Cipher Block Chaining mode, plaintext information is likely to begin to leak after $2^{b/2}$ blocks have been encrypted with the same key.

4. Column 4 indicates the size of the parameters associated with the standards that use discrete logs and finite field arithmetic (DSA as defined in ASC X9.30 for digital signatures, and Diffie-Hellman (DH) and MQV key agreement as defined in [ANS X9.42], where L is the size of the modulus p, and N is the size of q. L is commonly considered to be the key size for the algorithm, although L is actually the key size of the public key, and N is the key size of the private key.
5. Column 5 defines the value for k (the size of the modulus n) for the RSA algorithm specified in ANS X9.31 for digital signatures, and specified in ANS X9.44 for key establishment. The value of k is commonly considered to be the key size.
6. Column 6 defines the value of f (the size of n, where n is the order of the base point G) for the discrete log algorithms using elliptic curve arithmetic that are specified for digital signatures in ANS X9.62, and for key establishment as specified in ANS X9.63. The value of f is commonly considered to be the key size.

Table 7: Equivalent strengths.

Bits of security	Symmetric key algs.	Hash functions	DSA, D-H, MQV	RSA	Elliptic Curves
112	3-key TDEA	SHA-224	L = 2048 N = 224	k = 2048	$f \geq 224$
128	AES-128	SHA-256	L = 3072 N = 256	k = 3072	$f \geq 256$
192	AES-192	SHA-384			$f \geq 384$
256	AES-256	SHA-512			

C.3 Extracting Bits in the Dual_EC_DRBG (...)

C.3.1 Potential Bias Due to Modular Arithmetic for Curves Over F_p

For the mod p curves (i.e, a *Prime field curve*), there is a potential bias in the output due to the modular arithmetic. This behavior is succinctly explained in Part 1 of this Standard, and two approaches to correcting the bias are presented there. The Negligible Skew Method described in Section 14.2.2 of Part 1 is appropriate for the NIST curves, since all were selected to be over prime fields near a power of 2 in size. Each NIST prime has at least 32 leading 1's in its binary representation, and at least 16 of the leftmost (high-order) bits are discarded in each block produced. These two facts imply that there is a small fraction ($\leq 1/2^{32}$) of *outlen* outputs for which a bias to 0 may occur in one or more bits. This can only happen when the first 32 bits of an x-coordinate are all zero. As the leftmost 16 bits (at least) are discarded, an adversary can never be certain when a "biased" block has occurred. Thus, any bias due to the modular arithmetic may safely be ignored.

C.3.2 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 situation is slightly worse with the binary curves, since each has a cofactor of 2 or 4. This means that only about 1/4 or 1/8, respectively, of the m-bitstrings occur as x coordinates. Thus, the NIST elliptic curves have m-bit outputs that are lacking 1,2 or 3 bits of entropy, when taken in their entirety.

The "abouts" in the preceding example can be made more precise, taking into account the difference between 2^m and p, and the actual number of points on the curve (which is always within $2 * p^{½}$ of p). For the NIST curves, these differences won't matter at the scale of the results, so they will be ignored. This allows the heuristics given here to work for any curve with "about" $(2^m)/f$ points, where f = 1,2 or 4 is the curve's cofactor.

The basic assumption needed is that the approximately $(2^m)/(2f)$ x coordinates that do occur are "uniformly distributed": 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 $\{j=0\}$ to $\{j=2^d\}$ $[\ 2^{\{m-d\}}$ binomprob$(2^d, z, 2^d\text{-}j)]\ p_j \log_2\{p_j\}$.

The term in braces represents the approximate number of $(m\text{-}d)$-bitstrings, which fall into one of $1+2^d$ categories as determined by the number of times j it occurs in an x coordinate; $z = (2f\text{-}1)/2f$ 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

log2(*f*): 1 *d*: 0 entropy: 254.00000000 *m-d*: 256
log2(*f*): 1 *d*: 1 entropy: 253.75000000 *m-d*: 255
log2(*f*): 1 *d*: 2 entropy: 253.32398965 *m-d*: 254
log2(*f*): 1 d: 3 entropy: 252.68128674 m-d: 253
log2(*f*): 1 d: 4 entropy: 251.85475372 m-d: 252
log2(*f*): 1 *d*: 5 entropy: 250.93037696 *m-d*: 251
log2(*f*): 1 *d*: 6 entropy: 249.96572188 *m-d*: 250
log2(*f*): 1 *d*: 7 entropy: 248.98298045 *m-d*: 249
log2(*f*): 1 *d*: 8 entropy: 247.99151884 *m-d*: 248
log2(*f*): 1 *d*: 9 entropy: 246.99576643 *m-d*: 247
log2(*f*): 1 *d*: 10 entropy: 245.99788495 *m-d*: 246
log2(*f*): 1 *d*: 11 entropy: 244.99894291 *m-d*: 245
log2(*f*): 1 *d*: 12 entropy: 243.99947156 *m-d*: 244
log2(*f*): 1 *d*: 13 entropy: 242.99973581 *m-d*: 243
log2(*f*): 1 *d*: 14 entropy: 241.99986791 *m-d*: 242

log2(*f*): 1 *d*: 15 entropy: 240.99993397 *m-d*: 241

log2(*f*): 1 *d*: 16 entropy: 239.99996700 *m-d*: 240

log2(*f*): 2 *d*: 0 entropy: 253.00000000 *m-d*: 256

log2(*f*): 2 *d*: 1 entropy: 252.87500000 *m-d*: 255

log2(*f*): 2 *d*: 2 entropy: 252.64397615 *m-d*: 254

log2(*f*): 2 *d*: 3 entropy: 252.24578858 *m-d*: 253

log2(*f*): 2 *d*: 4 entropy: 251.63432894 *m-d*: 252

log2(*f*): 2 *d*: 5 entropy: 250.83126431 *m-d*: 251

log2(*f*): 2 *d*: 6 entropy: 249.91896704 *m-d*: 250

log2(*f*): 2 *d*: 7 entropy: 248.96005989 *m-d*: 249

log2(*f*): 2 *d*: 8 entropy: 247.98015668 *m-d*: 248

log2(*f*): 2 *d*: 9 entropy: 246.99010852 *m-d*: 247

log2(*f*): 2 *d*: 10 entropy: 245.99506164 *m-d*: 246

log2(*f*): 2 *d*: 11 entropy: 244.99753265 *m-d*: 245

log2(*f*): 2 *d*: 12 entropy: 243.99876678 *m-d*: 244

log2(*f*): 2 *d*: 13 entropy: 242.99938350 *m-d*: 243

log2(*f*): 2 *d*: 14 entropy: 241.99969178 *m-d*: 242

log2(*f*): 2 *d*: 15 entropy: 240.99984590 *m-d*: 241

log2(*f*): 2 *d*: 16 entropy: 239.99992298 *m-d*: 240

Observations:

a) Each table starts where it should, at 1, 2 or 3 missing bits;

b) The missing entropy rapidly decreases;

c) Each doubling of the log2(*f*)actor requires about 1 more bit to be discarded for the same level of entropy;

d) 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.

Based on these calculations, for the mod *p* curves, it is recommended that an implementation **shall** remove at least the **leftmost**, ie, most significant, 13 bits of every *m*-bit output, and that the **Dual_EC_DRBG (...)** be reseeded every 10,000 iterations. For the binary curves, either 14 or 15 of the leftmost bits **shall** be removed, as determined by the cofactor being 2 or 4, respectively. Using this value for *d* in the mod *p* curves insures that no bit has a bias from the modular reduction exceeding $1/2^{44}$

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

ANNEX D: (Informative) Functional Requirements

[Should this annex be retained? Should it just address those requirements that are appropriate for DRBGs?]

D.1 General Functional Requirements

The following functional requirements apply to all random bit generators:

1. *The implementation **shall** be designed to allow validation testing; including documenting specific design assertions about howt the RBG operates. This **shall** include mechanisms for testing all detectable error conditions.*

   Implementation validation testing for DRBGs is discussed in Section 11.3.

2. *The RBG **shall** be designed with the intent of meeting the security properties in Part 1, Section 8. This is on a best effort basis, as aspects of some of these properties are not testable.*

   *Documentation requirement: There **shall** be design documentation that describes how the RBG is intended to meet all security properties, including protection from misbehavior.*

   The fulfillment of general RBG requirements is discussed in Part 4. Part 1, Section 8 includes discussions of backtracking and prediction resistance, RBG output properties and RBG operational properties. Part 3-specific requirements are discussed below. Documentation requirements for RBGs are listed in Section 11.2.

3. *The RBG **shall** support backtracking resistance.* [I still think this is a wasted statement, since implied by requirement 2.]

   Backtracking resistance has been designed into each DRBG specified in Section 10.

Optional attributes for the functions in an RBG are as follows:

4. *The RBG **may** be capable of supporting prediction resistance.*

   An optional prediction resistance capability is specified for the DRBG functions in Section 9.2 - 9.4 and is also discussed in Section 8.6.

D.2 Functional Requirements for Entropy Input

These requirements are addressed in Parts 2 and 4 of this Standard.

D.3 Functional Requirements for Other Inputs

No general function requirements are stated in Part 1for other inputs. However, Part 3 requirements for other input are discussed in Section 7.2.3.

D.4 Functional Requirements for the Internal State

The requirements for the internal state of a RBG are:

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

   The internal state **shall** be protected at least as well as the intended use of the pseudorandom output bits requested by the consuming application. (see Section 8.2.3).

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

   This requirement is outside the scope of this Standard. Fulfilling this requirement may be addressed, for example, by implementing the DRBG in a FIPS 140-2 validated module. Further discussion of this requirement will be addressed in Part 4.

3. *The RBG **shall** satisfy the requirements for a particular security strength (from the set of [112, 128, 192, 256, or potentially unlimited]) in the internal state components.*

   *Documentation requirement: The security strength provided by the RBG **shall** be documented.*

   Sections 8.4, 9.2, 9.3 and the DRBG algorithms in Section 10 address the acquisition of sufficent entropy for the seed to satisfy a given security strength. Documentation requirements are listed in Section 11.2.

D.5 Functional Requirements for the Internal State Transition Function

The requirements for the internal state transition functions of an RBG are:

1. *The deterministic elements of internal state transition functions **shall** be verifiable via known-answer testing during installation and/or startup and/or initialization, and periodic health tests.*

   A DRBG **shall** perform self-tests to ensure that the DRBG continues to function properly. Self tests are discussed in Sections 9.7 and 11.4.

2. *The internal state transition function **shall**, over time, depend on all the entropy carried by the internal state. That is, added entropy **shall** affect the internal state.*

   This requirement is fulfilled by the design of the DRBGs specified in Section 10.

3. *The Internal State Transition Function **shall** resist observation and analysis via*

power consumption, timing, radiation emissions, or other side channels as appropriate, depending on the access by an observer who could be an adversary. What is appropriate (if anything) depends on the details of the implementation and shall be described by the implementation documentation.

*Documentation requirement: This aspect of the design **shall** be documented.*

This requirement is outside the scope of this Standard. Fulfilling this requirement may be addressed, for example, by implementing the DRBG in a FIPS 140-2 validated module. Part 4 will address this requirement further.

4. *It **shall not** be feasible (either intentionally or unintentionally) to cause the Internal State Transition Function to return to a prior state in normal operation (this excludes testing and authorized verification of the RBG output), except possibly by chance (depending on the specific design).*

   This requirement is fulfilled by the design of the DRBGs specified in Section 10.

D.6 Functional Requirements for the Output Generation Function

The functional requirements for the output generation function are:

1. *The output generation function **shall** be deterministic (given all inputs) and **shall** allow known-answer testing when requested.*

   The determinism of the output generation function is inherent in the DRBG algorithm designs of Section 10. Known answer testing is discussed in Sections 9.7, 11.3 and 11.4.

2. *The output **shall** be inhibited until the internal state obtains sufficient assessed entropy.*

   Section 8.4 states that a DRBG **shall not** provide output until a seed is available. Sections 9.2 - 9.5 request entropy at appropriate times during the instantiate, reseed and generate functions.

3. *Once a particular internal state has been used for output, the internal state **shall** be changed before more output is produced. The OGF **shall not** reuse any bit from the subset of bits of the pool that were used to produce output. An ISTF **shall** either update the internal state between successive actions of the OGF, or the OGF **shall** select independent subsets of bits in the internal state without reusing any previously selected bits between updates of the internal state by the ISTF. In the latter case, this process **shall** update the internal state in order to select a different set of bits from the "pool" of bits from which output is to be dervied.*

   *Documentation requirement: This aspect of the design **shall** be documented.*

   The specifications for the DRBG algorithms in Section 10 include an update of the internal state prior to returning the requested pseudorandom bits to the consuming application. Documentation requirements are listed in Section 11.2.

4. *Test output from a known answer test **shall** be separated from operational output (e.g., random output that is used for a cryptographic purpose).*

   Section 11.4.1 states that all data output from the DRBG module **shall** be inhibited while operational tests are performed. The results from known-answer tests **shall not** be output as random bits during normal operation.

5. *The output generation function **shall** protect the internal state, so that analysis of RBG outputs does not reveal useful information (from the point of view of compromise) about the internal state that could be used to reveal information about other outputs.*

   The DRBG algorithms specified in Section 10 have been designed to fulfill this requirement.

6. *The output generation function **shall** use information from the internal state that contains sufficient entropy to support the required security strength.*

   *Documentation requirement : This aspect of the design **shall** be documented.*

   Providing that the seed used to initialize the DRBG contains the appropriate amount of entropy for the required security strength, the output generation function in the DRBGs in this Standard have been designed to fulfill this requirement. Documentation requirements are listed in Section 11.2.

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

   *Documentation requirement: This aspect of the design **shall** be documented.*

   This requirement is outside the scope of this Standard. Fulfilling this requirement may be addressed, for example, by implementing the DRBG in a FIPS 140-2 validated module. Part 4 will discuss this requirement further.

D.7 Functional Requirements for Support Functions

The functional requirements for support functions in Part 1 are:

1. *An RBG **shall** be designed to permit testing that will ensure that the generator continues to operate correctly. These tests **shall** be performed at start-up (after either initialization or re-initialization), upon request and **may** also be performed periodically or continuously.*

   Section 11.4 specifies a requirement for operational (health) testing. A general method for operational testing is provided in Section 9.7.

2. *Output **shall** be inhibited during power-up, on-request and periodic testing until testing is complete and the result is acceptable. If the result is not acceptable, the RBG **shall** enter an error state.*

   Section 11.4 specifies that operational testing **shall** be conducted during power-up, on demand and at periodic intervals; this section also specifies that output **shall** be inhibited during testing. Section 9.7 specifies operational tests.

3. *Output need not be inhibited during continuous testing unless an unacceptable result is encountered. When an unacceptable result is thus determined, output **shall** be inhibited, and the RBG **shall** enter an error state.*

   Continuous testing is not specified for DRBGs.

4. *When an RBG fails a test, the RBG **shall** enter an error state and output an error indicator. The RBG **shall not** perform any operations while in the error state. The other parts of this Standard address error recovery in more detail, as appropriate.*

   Section 11.4 specifies this requirement. Sections 9.7 and 9.8 discuss the error handling process.

5. *Any other support functions implemented **shall** be documented regarding their purpose and the principles used in their design.*

   Documentation requirements are listed in Section 11.2.

ANNEX E: (Informative) DRBG Selection

[This will need to be revised, based on the DRBGs that are retained and the content of Part 4.]

E.1 Choosing a DRBG Algorithm

Almost no system designer starts with the idea that he's going to generate good random bits. Instead, he 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 him achieve that goal. Typically, as he begins to understand the requirements of those cryptographic mechanisms, he learns that he will also have to generate some random bits, and that this must be done with great care, or he may inadvertently weaken the cryptographic mechanisms that he has chosen to implement. At this point, there are two things that may guide the designer's choice of a DRBG:

a. He may already have decided to include a block cipher, hash function, keyed hash function, etc., as part of his implementation. By choosing a DRBG based on one of these mechanisms, 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 designer of a module that does RSA signatures probably already has available some kind of hashing engine, so one of the hash-based DRBGs 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 of these mechanisms, he can minimize the number of algorithms he has to trust.

   For example, a designer of a module that provides encryption with AES can implement an AES-based DRBG. Since the DRBG is based for its security on the strength of AES, the module's security is not made dependent on any additional cryptographic primitives or assumptions.

The DRBGs specified in this standard have different performance characteristics, implementation issues, and security assumptions.

E.2 DRBGs Based on Hash Functions

Two DRBGs are based on any Approved hash function: **Hash_DRBG,** and **HMAC_DRBG**. A hash function is composed of an initial value, a padding mechanism and a compression function; the compression function itself may be expressed as

Compress (I, X), where I is the initial value, and X is the compression function input. All of the cryptographic security of the hash function depends on the compression function, and the compression is by far the most time-consuming operation within the hash function.

The hash-based DRBGs in this Standard allow for some tradeoffs between performance, security assumptions required for the security of the DRBGs, and ease of implementation.

E.2.1 Hash_DRBG

Hash_DRBG is closely related to the DRBG specified in FIPS-186-2, and can be seen as an updated version of that DRBG that can be used as a general-purpose DRBG. Although a formal analysis of this DRBG is not available, it is clear that the security of the DRBG depends on the security of **Hashgen.** Specifically, an attacker can get a large number of sequences of values:

Hash (V), **Hash** (V+1), **Hash** (V+2), ...

If the attacker can distinguish any of these sequences from a random sequence of values, then the DRBG can be broken.

E.2.1.1 Implementation Issues

This DRBG requires a hash function, some surrounding logic, and the ability to add numbers modulo $2^{seedlen}$, where *seedlen* is the length of the seed. **Hash_DRBG** also uses **hash_df** internally when instantiating, reseeding, or processing additional input. Note that **hash_df** requires only access to a general-purpose hashing engine and the use of a 48-bit counter. The "critical state values" on which the **Hash_DRBG** depends for its security (V, C and *reseed_counter*) require *seedlen* + *outlen* + 48 bits of memory[6].

E.2.1.2 Performance Properties

Each time that **Hash_DRBG** is called, a compression function computation is required for each *outlen* bits of requested output (or portion thereof), where *outlen* is the size of the hash function output block. For example, if *outlen* = 160, and 360 bits of pseudorandom data are requested, three compression function calls are made (two to produce the first 320 bits, and a third from which to select the remaining 40 bits. In addition, there is a certain amount of overhead to updating the state in order to achieve backtracking resistance; this requires one compression function call and some additions modulo $2^{seedlenn}$, plus the update of *reseed_counter*. For the above example, a total of four compression function calls are required, three to generate the requested output bits, and one to update the state.

E.2.2 HMAC_DRBG

HMAC_DRBG is a DRBG whose security is based on the assumption that HMAC is a pseudorandom function. The security of **HMAC_DRBG** is based on an attacker getting sequences of up to 2^{35} bits, generated by the following steps:

temp = the *Null* string.

[6] V is *seedlen* bits long, C is *outlen* bits long (where *outlen* is the length of the hash function output block), and *reseed_counter* is a maximum of 48 bits in length.

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

V = **HMAC** (K, V).

temp = *temp* || V.

The steps in the "While" statement iterate $\lceil requested\_no\_of\_bits/outlen \rceil$ times. Intuitively, so long as V does not repeat, any algorithm that can distinguish this output sequence from an ideal random sequence can be used in a straightforward way to distinguish HMAC from a pseudorandom function.

Between these output sequences, both V and K are updated using the following steps (assuming no additional inputs):

K = **HMAC** (K, (V || 0x01)) = **Hash (opad** (K) || **Hash (ipad** (K) || (V || 0x01))).

V = **HMAC** (K, V) = **Hash (opad** (K) || **(Hash (ipad** (K) || V)).

where:

K and V are *outlen* bits long,

opad (K) is K exclusive-ored with (*inlen*/8) bytes of 0x5c, for a total of *inlen* bits,

ipad (K) is K exclusive-ored with (*inlen*/8) bytes of 0x36, for a total of *inlen* bits,

outlen is the length of the hash function output block, and

inlen is the length of the hash function input block.

E.2.2.1 Implementation Properties

The only thing required to implement this DRBG is access to a hashing engine. However, the performance of the implementation will improve enormously (by about a factor of two!) with either a dedicated **HMAC** engine, or direct access to the hash function's underlying compression function. The "critical state values" on which **HMAC_DRBG** depends for its security (K and V) take up 2**outlen* bits in the most compact form, but for reasonable performance, 3**outlen* bits are required in order to precompute padded values.

E.2.2.2 Performance Properties

HMAC_DRBG is about a factor of two slower than Hash_DRBG for long bitstrings produced by a single request. That is, each *outlen*-bit piece of the requested pseudorandom output requires two compression function calls to perform the **HMAC** computation. Each output request also incurs another six compression function calls to update the state.

Note that an implementation that has access only to a high-level hashing engine loses another factor of two in performance; if the performance of the DRBG is important, **HMAC_DRBG** requires either a dedicated **HMAC** engine or access to the compression function that underlies the hash function. However, if performance is not an important issue, the DRBG can be implemented using nothing but a high-level hashing engine.

E.2.3 Summary and Comparison of Hash-Based DRBGs

E.2.3.1 Security

It is interesting to contrast the two ways that the hash function is used in these DRBGs:

Hash_DRBG:

Hash (V), **Hash** (V+1), **Hash** (V+2)...

The only unknown input into the compression function used by the hash function is this sequence of secret values, $V+i$. Since the initial value of the hash function is publicly known, the adversary is given full knowledge of all but *seedlen* bits of input into the compression function, and knowledge of the close relationship between these inputs, as well.

HMAC_DRBG:

V_1 = **HMAC** (K, V_0) = **Hash (opad** (K) || **(Hash (ipad** (K) || V_0)).

V_2 = **HMAC** (K, V_1) = **Hash (opad** (K) || **(Hash (ipad** (K) || V_1)).

V_3 = **HMAC** (K, V_2) = **Hash (opad** (K) || **(Hash (ipad** (K) || V_2)).

etc

as specified in Annex E.2.2.

The adversary knows many specific bits of the input to the final compression function whose output he sees; for SHA-256, the compression function takes a total of 768 bits of input, and the adversary knows 256 of those bits[7]. (This is worse for SHA-1 and SHA-384.) On the other hand, the adversary doesn't even know the exclusive-or relationships for *outlen* bits of the message input. In the case of SHA-256, this means that 256 bits are unknown.

It is clear that **Hash_DRBG** makes stronger assumptions on the strength of the compression function, although they are not precisely comparable. Specifically, **HMAC_DRBG** allows an adversary to precisely know many bits of the input to the compression functions, but not to know complete exclusive-or or additive relationships between these bits of input.

[7] The innermost hash function provides *outlen* bits of input after its two compression function calls on **ipad** (K) and V. The outermost hash function also requires two compression functions: the first operates on **opad** (K) and produces *outlen* bits that are used as the chaining value for the final compression function on the result from the innermost hash function concatenated with the hash function padding. Therefore, the input to the final compression function is the length of the chaining value (*outlen* bits) + the length of the ouput from the innermost hash function (*outlen* bits) + the length of the padding (*inlen* - *outlen* bits). In the case of SHA-256, where *inlen* = 512, and *outlen* = 256, the length of the input to the last compression function is 768 bits, of which only the padding bits are known (256 bits).

E.2.3.2 Performance / Implementation Tradeoffs

The following performance and implementation tradeoffs should be considered when selecting a hash-based DRBG with regard to the overhead associated with requesting pseudorandom bits, the cost of actually generating *outlen* bits (not including the overhead), and the memory required for the critical state values for each DRBG. The overhead is, essentially, the cost of updating the state prior to the next request for pseudorandom bits. The cost of generating each *outlen* block of bits of output should be multiplied by the number of *outlen*-bit blocks of output required in order to obtain the true cost of pseudorandom bit generation. Both the overhead and generation costs assume that prediction resistance and reseeding are not required, and that additional input is not provided for the request: if this is not the case, the costs are increased accordingly. Note that the memory requirements do not take into account other information in the state that is required for a given DRBG.

Hash_DRBG:

Request overhead: one compression function and several additions mod $2^{seedlen}$.

Cost for *outlen* bits of pseudorandom output: one compression function.

Memory required for the critical state values *V*, *C* and *reseed_counter*: *inlen* + *outlen* + 32 bits.

HMAC_DRBG (with access to the hash function's compression function):

Request overhead: six compression functions[8].

Cost for *outlen* bits of pseudorandom output: two compression functions.

Memory required for the critical state values *K* and *V*: 3**outlen* bits when precomputation is used .

HMAC_DRBG (hash engine access only):

Request overhead: eight compression function calls[9].

Cost for *outlen* bits of pseudorandom output: four compression functions[10].

Memory required for the critical state values *K* and *V*: 2**outlen* bits, since precomputation is unavailable.

For these DRBGs, additional inputs provided during pseudorandom bit generation add considerably to the request overhead. Instantiation and reseeding are somewhat more expensive than pseudorandom output generation; however, these relatively rare operations can afford to be somewhat more expensive to minimize the chances of a successful attack.

---

[8] Two compression functions for each HMAC computation, and two compression functions for precomputation.

[9] There are two HMAC computations, each requiring two hash function calls. Each hash computation requires two compression function calls.

[10] The single HMAC computation requires four compression functions as explained in the previous footnote.

E.3 DRBGs Based on Block Ciphers

E.3.1 The Two Constructions: CTR and OFB

This standard describes two classes of DRBGs based on block ciphers: One class uses the block cipher in OFB-mode, the other class uses the CTR-mode. There are no practical security differences between these two DRBGs; CTR mode guarantees that short cycles cannot occur in a single output request, while OFB-mode guarantees that short cycles will have an extremely low probability. OFB-mode makes slightly less demanding assumptions on the block cipher, but the security of both DRBGs relates in a very simple and clean way to the security of the block cipher in its intended applications. This is a fundamental difference between these DRBGs and the DRBGs based on hash functions, where the DRBG's security is ultimately based on pseudorandomness properties that do not form a normal part of the requirements for hash functions. An attack on any of the hash-based DRBGs does not necessarily represent a weakness in the hash function; however, for these block cipher-based constructions, a weakness in the DRBG is directly related to a weakness in the block cipher.

Specifically, suppose that there is an algorithm for distinguishing the outputs of either DRBG from random with some advantage. If that algorithm exists, it can be used to build a new algorithm for distinguishing the block cipher from a random permutation, with the same time and memory requirements and advantage.

Because there is no practical security difference between the two classes of block-cipher based DRBGs, the choice between the two constructions is entirely a matter of implementation convenience and performance. An implementation that uses a block cipher in OFB, CBC, or full-block CFB mode can easily be used to implement the OFB-based DRBG construction; an implementation that already supports counter mode can reuse that hardware or software to implement the counter-mode DRBG. In terms of performance, the CTR-mode construction is more amenable to pipelining and parallelism, while the OFB-mode construction seems to require slightly less supporting hardware.

E.3.2 Choosing a Block Cipher

While security is not an issue in choosing between the two DRBG constructions, the choice of the block cipher algorithm to be used is more of an issue. At present, only TDEA and AES are approved block cipher algorithms. ~~However, the two block cipher DRBG constructions will work for any block cipher with a block length ≥ 64 and key length ≥ 112. TDEA's 64-bit block imposes some fundamental limits on the security of these constructions, though these limits don't appear to lead to practical security issues for most applications.~~

Consider a sequence of the maximum permitted number of generate requests, each producing the maximum number of DRBG outputs from each generate call. Assuming that the block cipher behaves like a pseudorandom permutation family, the probability of distinguishing the full sequence of output bytes is:

1. For AES-128, there are a maximum of 2^{28} blocks (i.e., 2^{32} bytes = 2^{35} bits) generated per **Generate (...)** request, 2^{32} total **Generate (...)** requests allowed, 2^{128} possible keys, and 2^{128} possible starting blocks.
   a. The probability of an internal collision in a single **Generate (...)** request is never higher than about 2^{-96}, and so the probability of an internal collision in any given **Generate (...)** request is never higher than about 2^{-64}. (This applies only to the OFB-mode, but a collision of this kind would result in a very easy distinguisher.)
   b. The expected probability of an internal collision in a sequence of 2^{28} random 128-bit blocks is about 2^{-74}. Thus, the probability of seeing an internal collision in any of the **Generate (...)** sequences is about 2^{-42}. This probability is low enough that it does not provide an efficient way to distinguish between DRBG outputs and ideal random outputs.
   c. The probability of a key colliding between any two **Generate (...)** requests in the sequence of 2^{32} such requests is never larger than about 2^{-65}. This is also negligible. (For AES-192 and AES-256, this probability is even smaller.)
2. For three-key TDEA with 168-bit keys and 64-bit blocks, things are a bit different: There are 2^{16} **Generate (...)** requests allowed, and a maximum of 2^{13} blocks (i.e., 2^{16} bytes = 2^{19} bits) generated per **Generate (...)** request. (Note that this breaks the more general model in this document of assuming 2^{64} innocent operations.) In this case:
   a. The probability of an internal collision is never higher than about 2^{-51} per **Generate (...)** request, and with only 2^{16} such requests allowed, the probability of ever seeing such an internal collision in a sequence of requests is never more than about 2^{-35}. (Note that if more requests are allowed, as required by the 2^{64} bound assumed elsewhere in the document, there would be an unacceptably high probability of this event happening at least once.)
   b. The expected probability of an internal collision in a sequence of 2^{13} 64-bit blocks is about 2^{-38}. Thus, the probability of ever seeing an internal collision in 2^{16} output sequences is still an acceptably low 2^{-22}. (Note that if more **Generate (...)** requests are allowed, there would be an unacceptably high probability of this happening, leading to an efficient distinguisher between this DRBG's outputs and ideal random outputs.
   c. The probability of a key colliding between any two of the 2^{16} **Generate (...)**

requests is about 2^{-136}, which is negligible.

To summarize: block size matters much more than the choice of DRBG construction that is used. The limits on the numbers of **Generate** (...) requests and the number of output bits per request require frequent reseeding of the DRBG. Furthermore, the limits guarantee that even with reseeding, an adversary that is given a really long sequence of DRBG outputs from several reseedings cannot distinguish that output sequence from random reliably. The block cipher DRBGs used with TDEA are suitable for low-throughput applications, but not for applications requiring really large numbers of DRBG outputs. **For concreteness, if an application is going to require more than 2^{32} output bytes (2^{35} bits) in its lifetime, that application should not use a block cipher DRBG with TDEA ~~or any other 64-bit block cipher~~.**

E.3.3 Conditioned Entropy Sources and the Derivation Function

[Some or all of this section probably belongs in Part 4]

The block cipher DRBGs are defined to be used in one of two ways for initializing the DRBG state during instantiation and reseeding: Either with freeform input strings containing some specified amount of entropy, or with full-entropy strings of precisely specified lengths The freeform strings will require the use of a derivation function, whereas the use of full-entropy strings will not. The block cipher derivation function uses the block cipher algorithm to compute several parallel CBC-MACs on the input string under a fixed key and using different IVs, uses the result to produce a key and starting block, and runs the block cipher in OFB-mode to generate outputs from the derivation function. An implementation must choose whether to provide full entropy, or to support the derivation function. This is a high-level system design decision; it affects the kinds of entropy sources that may be used, the gate count or code size of the implementation, and the interface that applications will have to the DRBG. On one extreme, a very low gate count design may use ~~hardware~~ entropy sources that are easily conditioned, such as a bank of ring oscillators that are exclusive-ored together, rather than to support a lot of complicated processing on input strings. On the other extreme, a general-purpose DRBG implementation may need the ability to process freeform input strings as personalization strings and additional inputs; in this case, the block cipher derivation function must be implemented.

E.4 DRBGs Based on Hard Problems

The **Dual_EC_DRBG** and **MS_DRBG** base their security on a "hard" number-theoretic problem. For the types of curves used in the **Dual_EC_DRBG**, the Elliptic Curve Discrete Logarithm Problem has no known attacks that are better than the "meet-in-the-middle" attacks, with a work factor of sqrt (2^m). In the case of MS_DRBG, which is based loosely on the RSA problem, the work factor of the best algorithm is more complex to state, but well-established.

These algorithms are decidedly less efficient to implement than some of the others. However, in those cases where security is the utmost concern, as in SSL or IKE exchanges, the additional complexity is not usually an issue. Except for dedicated servers, time spent on the exchanges is just a small portion of the computational load; overall, there is no impact on throughput by using a number-theoretic algorithm. As for SSL or IPSEC servers, more and more of these servers are getting hardware support for cryptographic primitives like modular exponentiation and elliptic curve arithmetic for the protocols themselves. Thus, it makes sense to utilize those same primitives (in hardware or software) for the sake of high-security random numbers.

E.4.1 Implementation Considerations

E.4.1.1 Dual_EC_DRBG

Random bits are produced in blocks of bits representing the x-coordinates on an elliptic curve.

Because of the various security levels allowed by this Standard there are multiple curves available , with differing block sizes. The size is always a multiple of 8, about 16 bits less than a curve's underlying field size. Blocks are concatenated and then truncated, if necessary, to fullfil a request for any number of bits up to a maximum per call of 10,000 times the block length. The smallest blocksize is 216, meaning that at least 2M bits can be requested on each call.)

An important detail concerning the Dual_EC_DRBG is that every call for random bits, whether it be for 2 million bits or a single bit, requires that at least one full block of bits be produced; no unused bits are saved internally from the previous call. Each block produced requires two point multiplications on an elliptic curve—a fair amount of computation. Applications such as IKE and SSL are encouraged to aggregate all their needs for random bits into a single call to Dual_EC_DRBG, and then parcel out the bits as required during the protocol exchange. A C structure, for example, is an ideal vehicle for this.

Comment [ebb10]: Page: 132
Doesn't this violate our guidance somewhere ?

To avoid unnecessarily complex implementations, it should be noted that *every* curve in the Standard need not be available to an application. For instance, one may choose to do arithmetic only over the prime order fields in a software application, or perhaps a particular binary curve in a hardware application. To improve efficiency, there has been much research done on the implementation of elliptic curve arithmetic; descriptions and source code are available in the open literature.

As a final comment on the implementation of the Dual_EC_DRBG, note that having fixed base points offers a distinct advantage for optimization. Tables can be precomputed that allow nP to be attained as a series of point additions, resulting in an 8 to 10-fold speedup, or more, if space permits.

E.4.1.2. Micali-Schnorr

Micali-Schnorr was designed to be a more efficient version of the predecessor algorithm, the Blum-Blum-Shub (BBS) DRBG. BBS uses the recursion $x_i = x_{i-1}^2 \bmod n$ to generate its state sequence, producing a single pseudorandom bit as the least significant bit of x_i. Later, it was shown that O(ln(ln n))) bits could be taken on each iteration, but this is still a very small percentage of those produced. The MS_DRBG allows a much larger percentage of n bits to be used on each iteration, and has an additional advantage in that no output bits are used to propagate the sequence. It does, however, rely on a stronger assumption for its security than the intractability of integer factorization.

As ANS X9.82 standard evolved, committee members argued for restricting the number of bits generated on each exponentiation to O(ln(ln n)) *hard* bits, as is done in BBS. The result is that the efficiency argument for choosing MS over BBS doesn't apply. Nonetheless, a user does have more options in the choice of parameters.

Micali_Schnorr offers an alternative to Dual_EC_DRBG in the class of algorithms based on a hard problem from number theory, and presents an advantage in its simplicity. All that's required for implementation is a routine that computes x^e mod n; this can be readily found in commercial and open source toolkits.

ANNEX F: (Informative) Example Pseudocode for Each DRBG

[These examples do not reflect the latest changes to Part 3. They will be revised when the decision is made as to which DRBGs will be retained.]

F.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*.

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.

F.2 Hash_DRBG Example

F.2.1 Discussion

This example of **Hash_DRBG** uses the SHA-1 hash function, and prediction resistance is supported in the example. Both a personalization string and additional input are allowed. A 32-bit incrementing counter is used as the nonce for instantiation (*instantiation_nonce*); 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.

A total of 10 internal states are provided (i.e., 10 instantiations may be handled simultaneously).

For this implementation, the functions and algorithms are "inline", i.e., the algorithms are not called as separate routines from the function envelopes.

The internal state contains values for *V*, *C*, *reseed_counter*, *security_strength* and *prediction_resistance_flag*, where *V* and C are bitstrings, and *reseed_counter*, *security_strength* and the *prediction_resistance_flag* are integers. A requested prediction resistance capability is indicated when *prediction_resistance_flag* = 1. Note: an empty internal state is represented as {*Null*, *Null*, 0, 0, 0}.

In accordance with Table 3 in Section 10.1.1, the 112 and 128 bit security strengths may be supported. Using SHA-1, the following definitions are applicable for the instantiate, generate and reseed functions and algorithms:

1. *highest_supported_security_strength* = 128.
2. Output block length (*outlen*) = 160.
3. Required minimum entropy for instantiation and reseed = *security_strength.*
4. Minimum entropy input length (*min _length*) = *security_strength.*
5. Seed length (*seedlen*) = 440.
6. Maximum number of bits per request (*max_number_of_bits_per_request*) = 5000 bits.
7. Reseed interval (*reseed_interval*) = 100,000 requests.
8. Maximum length of the personalization string (*max_personalization_string_length*) = 500 bits.
9. Maximum length of additional_input (*max_additional_input_string_length*) = 500 bits.
10. Maximum length of entropy input (*max _length*) = 1000.

F.2.2 Instantiation of Hash_DRBG

This implementation will return a text message and an invalid state handle (-1) when an error is encountered. Note that the value of *instantiation_nonce* is an internal value that is always available to the instantiate function.

Note that this implementation does not check the *prediction_resistance_flag*, since the implementation can handle prediction resistance. However, if an application actually wants prediction resistance, the implementation expects that *prediction_resistance_flag* = 1 during instantiation; this will be used in the generate function in Annex F.2.4.

Instantiate_Hash_DRBG (...):

Input: integer (*requested_instantiation_security_strength, prediction_resistance_flag*), bitstring *personalization_string*).

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

Process:

Comment: Check the input parameters.

1. If (*requested_instantiation_security_strength* > 128), then **Return** ("Invalid *requested_instantiation_security_strength*", -1).
2. If (**len** (*personalization_string*) > 500), then **Return** ("*Personalization_string* too long", -1).

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

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** (*security_strength*, *security_strength*, 1000).

5. If (*status* ≠ "Success"), then **Return** ("Failure indication returned by the *entropy_input* source:" || *status*, -1).

   Comment: Increment the nonce; actual coding must ensure that it wraps when it's storage limit is reached.

6. *instantiation_nonce* = *instantiation_nonce* + 1.

   Comment: The instantiate algorithm is provided in steps 7-11.

7. *seed_material* = *entropy_input* || *instantiation_nonce* || *personalization_string*.

8. *seed* = **Hash_df** (*seed_material*, 440).

9. V = *seed*.

10. C = **Hash_df** ((0x00 || V), 440).

11. *reseed_counter* = 1.

    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*) = {V, C, *reseed_counter*, *security_strength*, *prediction_resistance_flag*}.

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

F.2.3 Reseeding a Hash_DRBG Instantiation

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

Reseed_Hash_DRBG_Instantiation (...):

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

Output: string *status*.

Process:

Comment: Check the validity of the *state_handle*.

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

Comment: Get the internal state values needed to determine the new internal state.

2. Get the appropriate *internal_state* values, e.g., *V* = *internal_state*(*state_handle*).*V*, *security_strength* = *internal_state*(*state_handle*).*security_strength*.

Check the length of the *additional_input*.

3. If (**len** (*additional_input*) > 500), then **Return** ("*Additional_input* too long").

Comment: Get the *entropy_input*.

4. (*status, entropy_input*) = **Get_entropy** (*security_strength*, *security_strength*, 1000).
5. If (*status* ≠ "Success"), then **Return** ("Failure indication returned by the *entropy_input* source:" || *status*).

Comment: The reseed algorithm is provided in steps 7-11.

6. *seed_material* = 0x01 || *V* || *entropy_input* || *additional_input*.
7. *seed* = **Hash_df** (*seed_material*, 440).
8. *V* = *seed*.
9. *C* = **Hash_df** ((0x00 || *V*), 440).
10. *reseed_counter* = 1.

Comment: Update the *working_state* portion of the internal state.

11. Update the appropriate *state* values.
    11.1 *internal_state* (*state_handle*).*V* = *V*.
    11.2 *internal_* state (*state_handle*).*C* = *C*.
    11.3 *internal_ state* (*state_handle.reseed_counter* = *reseed_counter*.
12. **Return** ("Success").

F.2.4 Generating Pseudorandom Bits Using Hash_DRBG

The implementation returns a *Null* string as the pseudorandom bits if an error has been detected. Prediction resistance is requested when *prediction_resistance_request* = 1.

In this implementation, prediction resistance is requested by supplying *prediction_resistance_request* = 1 when the **Hash_DRBG** function is invoked.

Hash_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 the *state_handle*.

1. If ((*state_handle* > 9) or (*state* (*state_handle*) = {*Null*, *Null*, 0, 0, 0})), then **Return** ("State not available for the *state_handle*", *Null*).

   Comment: Get the internal state values.

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

   Comment: Check the validity of the other input parameters.

3. If (*requested_no_of_bits* > 5000) 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*) > 500), then **Return** ("*Additional_input* too long", *Null*).

6. If ((*prediction_resistance_request* = 1) and (*prediction_resistance_flag* ≠ 1)), then **Return** ("Prediction resistance capability not instantiated", *Null*).

   Comment: Reseed if necessary. Note that since the instantiate algorithm is inline with the functions, this step has been written as a combination of steps 6 and 7 of Section 9.4 and step 1 of the generate algorithm in Section 10.1.2.2.4. Because of this combined step, step 11.4 of Section 7.4.is not required.

7. If ((*reseed_counter* > 100,000) OR (*prediction_resistance_request* = 1)), then

    7.1 *status* = **Reseed_ Hash_DRBG_Instantiation** (*state_handle, additional_input*).

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

    Comment: Get the new internal state values.

    7.3 *V* = *internal_state* (*state_handle*).*V*, *C* = *internal_state* (*state_handle*).*C*, *reseed_counter* = *internal_state* (*state_handle*).*reseed_counter*, *security_strength* = *internal_state* (*state_handle*).*security_strength*, *prediction_resistance_flag* = *internal_state* (*state_handle*).*prediction_resistance_flag*.

    7.4 *additional_input* = *Null*.

    Comment: Steps 8-16 provide the rest of the generate algorithm. Note that in this implementation, the **Hashgen** routine is also inline as steps 9-13.

8. If (*additional_input* ≠ *Null*), then do

    7.1 w = **Hash** (0x02 || V || *additional_input*).

    7.2 $V = (V + w) \bmod 2^{440}$.

9. $m = \left\lceil \frac{requested\_no\_of\_bits}{outlen} \right\rceil$.

10. *data* = V.

11. W = the Null string.

12. For i = 1 to m

    12.1 w_i = **Hash** (*data*).

    12.2 $W = W \| w_i$.

    12.3 $data = (data + 1) \bmod 2^{seedlen}$.

13. *pseudorandom_bits* = Leftmost (*requested_no_of_bits*) bits of W.

14. H = **Hash** (0x03 || V).

15. $V = (V + H + C + reseed\_counter) \bmod 2^{440}$.

16. *reseed_counter* = *reseed_counter* + 1.

    Comments: Update the *working_state*.

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

    13.1 *internal_state* (*state_handle*).$V = V$.

    13.2 *internal_state* (*state_handle*).*reseed_counter* = *reseed_counter*.

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

F.3 HMAC_DRBG Example

F.3.1 Discussion

This example of HMAC_DRBG uses the SHA-256 hash function. The reseed and, thus, the prediction resistance is not provided. The nonce for intantiation consists of a random value with 64-bits of entropy; the nonce is obtained by increasing the call for entropy bits via the **Get_entropy** call by 64 bits (i.e., by adding 64 bits to the *security_strength* value).

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.

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 3 in Section 10.1.1, security strengths of 112, 128, 192 and 256 may 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.
3. Required minimum entropy for instantiation = *security_strength* + 64 (this includes the entropy required for the nonce).
4. Minimum entropy input length (*min _length*) = *security_strength* + 64 (this includes the minimum length for the nonce).
5. Seed length (*seedlen*) = 440.
6. Maximum number of bits per request (*max_number_of_bits_per_request*) = 7500 bits.
7. Reseed_interval (*reseed_ interval*) = 10,000 requests.
8. Maximum length of the personalization string (*max_personalization_string_length*) = 100.
9. Maximum length of the entropy input (*max _length*) = 1000.

F.3.2 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*)>100), 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_inptu and the nonce*.

4. *min_entropy* = *security_strength* + 64.
5. (*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_entropy*, 1000).
6. If (*status* ≠ "Success"), then **Return** ("Failure indication returned by 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* = 0.
6. **Return** (*V, Key, reseed_counter*).

F.3.3 Generating Pseudorandom Bits Using HMAC_DRBG

The implementation returns a *Null* string as the pseudorandom bits if an error has been detected. Ths function uses the **Update** function specified in Section 10.1.3.2.2.

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* > 3) 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.

6. (*status*, *pseudorandom_bits*, *V*, *Key*, *reseed_counter*) = **Generate_algorithm** (*V*, *Key*, *reseed_counter*, *requested_number_of_bits*).

7. If (*status* ≠ "Success"), then **Return** ("DRBG can no longer be used. Please re-instantiate or reseed", *Null*).

   Comment: Update the internal state.

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

12. **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*).

F.4 CTR_DRBG Example

F.4.1 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 uses AES-128 in the ECB mode.

The nonce for instantiation (*instantiation_nonce*) consists of a 32-bit incrementing counter (*instantiation_counter*) appended to 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*, *security_strength* and *prediction_resistance_flag*, where *V* and *Key* are integers, and all other values are integers.

In accordance with Table 4 in Section 10.2.1, security strengths of 112 and 128 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.
3. Key length (*keylen*) = 128.
4. Required minimum entropy for instantiate and reseed = *security_strength.*
5. Minimum entropy input length (*min _length*) = *security_strength.*
6. Maximum entropy input length (*max _length*) = 1000.
7. Maximum personalization string input length (*max_personalization_string_input_length*) = 500.
8. Maximum additional input length (*max_additional_input_length*) = 500.
9. Seed length (*seedlen*) = 256.
10. Maximum number of bits per request (*max_number_of_bits_per_request*) = 4000.
11. Reseed_interval (*reseed_interval*) = 100,000 requests.

F.4.2 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*).

F.4.3 Instantiation of CTR_DRBG

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 9.6.3, and uses AES-128 in ECB mode as the **Block_Encrypt** function.

Note that this implementation does not check the *prediction_resistance_flag*, since the implementation can provide prediction resistance. However, if an application actually wants prediction resistance for a pseudorandom bitstring, the implementation expects that *prediction_resistance_flag* = 1 during instantiation (i.e., an application may not require prediction resistance for an instantiation).

Instantiate_CTR_DRBG (...):

Input: integer (*requested_instantiation_security_strength, prediction_resistance_flag*), bitstring *personalization_string*.

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

Process:

Comment: Check the validity of the input parameters.

1. If (*requested_instantiatio_security_strength* > 128) then **Return** ("Invalid *requested_instantiation_security_strength*", -1).
2. If (**len** (*personalization_string*) > 500), 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** (*security_strength, security_strength*, 1000).
5. If (*status* ≠ "Success"), then **Return** ("Failure indication returned by the entropy source" || *status*, -1).

   Comment: Increment the nonce; actual coding must ensure that it wraps when it's storage limit is reached.

6. *instantiation_counter* = *instantiation_counter* + 1.
7. *instantiation_nonce* = *personalization_string* || *instantiation_counter*.

   Comment: Invoke the instantiate algorithm.

8. (*V, Key, reseed_counter*) = **Instantiate_algorithm** (*entropy_input*,

instantiation_nonce, *personalization_string*).

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

9. (*status*, *state_handle*) = **Find_state_space** ( ).

10. If (*status* ≠ "Success"), then **Return** ("No available state space" || *status*, -1).

Comment: Save the internal state.

11. *internal_state_* (*state_handle*) = { *V*, *Key*, *reseed_counter*, *security_strength*, *prediction_resistance_flag* }.

12. **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*).

F.4.4 Reseeding a CTR_DRBG Instantiation

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* > 5) or (*internal_state*(*state_handle*) = {*Null*, *Null*, 0, 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*, *prediction_resistance_flag* = *internal_state* (*state_handle*).*prediction_resistance_flag*.
3. If (**len** (*additional_input*) > 500), then **Return** ("*Additional_input* too long").
4. *min_entropy* = *security_strength* + 64.
5. (*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_entropy*, 1000).
6. If (*status* ≠ "Success"), then **Return** ("Failure indication returned by the entropy source" || *status*).

   Comment: Invoke the reseed algorithm.
7. (*V*, *Key*, *reseed_counter*) = **Reseed_algorithm** (*V*, *Key*, *reseed_counter*, *entropy_input*, *additional_input*).

   Comment: Save the new internal state.
8. *internal_state* (*state_handle*) = {*V*, *Key*, *reseed_counter*, *security_strength*, *reseed_counter*, *prediction_resistance_flag*}.
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*).

F.4.5 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* > 5) or (*internal_state* (*state_handle*) = {*Null*, *Null*, 0, 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*, *prediction_resistance_flag* = *internal_state* (*state_handle*).*prediction_resistance_flag*.

   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*) > 500), then **Return** ("*Additional_input* too long", *Null*).
6. If ((*prediction_resistance_request* = 1) and (*prediction_resistance_flag* ≠ 1)), then **Return** ("Prediction resistance capability not instantiated", *Null*).
7. *reseed_required_flag* = 0.
8. If (*reseed_required_flag* = 1) or (*prediction_resistance_request* = 1)), then

   8.1 *status* = **Reseed_CTR_DRBG_Instantiation** (*state_handle*, *additional_input*).

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

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

   8.3 *V* = *internal_state* (*state_handle*).*V*, *Key* = *internal_state* (*state_handle*).*Key*, *reseed_counter* = *internal_state* (*state_handle*).*reseed_counter*.

   8.4 *additional_input* = *Null*.

   8.5 *reseed_request_flag* = 0.

Comment: Generate bits using the generate algorithm.

9. (*status, pseudorandom_bits, V, Key, reseed_counter*) = **Generate_algorithm** (*V, Key, reseed_counter, requested_number_of_bits, additional_input*).

10. If (*status* ≠ "Success"), then

    10.1 *reseed_required_flag* = 1.

    10.2 Go to step 8.

Comment: Collect bits.

11. *internal_state* (*state_handle*) = {*V, Key, security_strength, reseed_counter, prediction_resistance_flag*}.

Comment: Determine the pseudorandom bits to be returned.

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

Generate_algorithm (...):

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

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

Process:

1. If (*reseed_counter* > 100,000), then **Return** ("Failure", *Null, V*, Key, *reseed_counter*).

2. If (*additional_input* ≠ *Null*), then

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

    2.2 If (*temp* > 256), then *additional_input* = **Block_Cipher_df** (*additional_input*, 256).

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

    2.4 (*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*).

F.5 OFB_DRBG Example

F.5.1 Discussion

This example of **OFB_DRBG** uses 3 key TDEA. Full entropy is available, and a block cipher derivation function is not used ; therefore, a nonce is not used. Prediction resistance is supported. A total of 5 internal states are available. A personalization string is allowed during instantiation, and additional input is allowed during reseeding and a request for pseudorandom bit generation. For this implementation, the functions and algorithms are written as separate routines. The **Block_Encrypt** function uses 3 key TDEA in the ECB mode.

The internal state contains the values for *V*, *Key*, *reseed_counter*, *security_strength* and *prediction_resistance_flag*; *V* and *Key* are integers; *reseed_counter*, *security_strength* and *prediction_resistance_flag* are integers.

In accordance with Table 4 in Section 10.2.1, a security strength of 112 is supported. Using 3 key TDEA, the following definitions are applicable for the instantiate, reseed and generate functions:

1. *highest_supported_security_strength* = 112.
2. Output block length (*outlen*) = 64.
3. Key length (*keylen*) = 168.
4. Number of bits for entropy input if full entropy is supported and a derivation function is not used: 232.
5. Minimum entropy input length (*min _length*) = *min_entropy* = 232.
6. Maximum entropy input length (*max _length*) = 232.
7. Maximum personalization string input length (*max_personalization_string_input_length*) = 232.
8. Maximum additional input length (*max_additional_input_length*) = 232.
9. Seed length (*seedlen*) = 232.
10. Maximum number of bits per request (*max_number_of_bits_per_request*) = 1000.
12. Reseed interval (*reseed_interval*) = 10,000 requests.

F.5.2 The Update Function

Update (...):

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

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

Process:

1. *temp* = *Null*.
2. While (**len** (*temp*) < 232) do
   2.1 *V* = **TDEA_ECB Encrypt** (*Key*, *V*).
   2.2 *temp* = *temp* || *V*.
3. *temp* = Leftmost 232 bits of *temp*.
4. *temp* = *temp* ⊕ *provided_data*.
5. *Key* = Leftmost 168 bits of *temp*.
6. *V* = Rightmost 64 bits of *temp*.
7. **Return** (*Key*, *V*).

F.5.3 Instantiation of OFB_DRBG

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

Note that this implementation does not use the *prediction_resistance_flag*, since it is known that prediction resistance is supported. However, if *prediction_resistance_flag* = 1, then a prediction resistance capability is requested for the instantiation.

Instantiate_OFB_DRBG (...):

Input: integer (*requested_instantiation_security_strength*, *prediction_resistance_flag*), bitstring *personalization_string*.

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

Process:

Comment: Check the validity of the input parameters.

1. If (*requested_instantiation_security_strength* > 112) then **Return** ("Invalid *requested_instantiation_security_strength*", -1).
2. If (**len** (*personalization_string*) > 232), then **Return** ("*Personalization_string* too long", -1).
3. *security_strength* = 112.

Comment: Get the entropy input.

4. (*status*, *entropy_input*) = **Get_entropy** (232, 232, 232).
5. If (*status* ≠ "Success"), then **Return** ("Failure indication returned by the entropy source" || *status*, -1).

Comment: Invoke the instantiate algorithm.

6. (*V*, *Key*, *reseed_counter*) = **Instantiate_algorithm** (*entropy_input*, *personalization_string*).
7. (*status*, *state_handle*) = **Find_state_space** ( ).
8. If (*status* ≠ "Success"), then **Return** ("No available state space" || *status*, -1).

Comment: Save the internal state.

9. *internal_state_* (*state_handle*) = {*V*, *Key*, *reseed_counter*, *security_strength*, *prediction_resistance_flag*}.
10. **Return** ("Success", *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. $Key = 0^{168}$. Comment: 168 bits.
3. $V = 0^{64}$. Comment: 64 bits.
4. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).
5. *reseed_counter* = 1.
6. **Return** ("Success", *V*, *Key*, *reseed_counter*).

F.5.4 Reseeding the OFB_DRBG Instantiation

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

Reseed_OFB_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* > 5) or (*internal_state* (*state_handle*)= {*Null*, *Null*, 0, 0}), then **Return** ("State not available for the indicated *state_handle*").

   Comment: Get the necessary 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*) > 232), then **Return** ("*Additional_input* too long").

   Comment: Get the *entropy_input*.

4. (*status*, *entropy_input*) = **Get_entropy** (232, 232, 232).
5. If (*status* ≠ "Success"), then **Return** ("Failure indication returned by the entropy source" || *status*).

   Comment: Invoke the reseed algorithm.

6. (*V*, *Key*, *reseed_counter*) = **Reseed_algorithm** (*V*, *Key*, *entropy_input*, *additional_input*).
7. *internal_state* (*state_handle*).*V* = *V*; *internal_state* (*state_handle*).*Key* = *Key*; *internal_state* (*state_handle*).*reseed_counter* = *reseed_counter*.
8. **Return** ("Success").

Reseed_algorithm (...):

Input: bitstring (*V*, *Key*), bitstring (*entropy_input*, *additional_input*).

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

Process:

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

   Comment: If the *additional_input* < 232, pad with zeros.

2. If (*temp* < 232), then *additional_input* = *additional_input* || $0^{232 - temp}$.
3. *seed_material* = *entropy_input* ⊕ *additional_input*.
4. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).
5. *reseed_counter* = 1.
6. **Return** (*V*, *Key*, *reseed_counter*).

F.5.5 Generating Pseudorandom Bits using OFB_DRBG

The implementation returns a *Null* string as the pseudorandom bits if an error has been detected. Note that prediction resistance is requested when *prediction_resistance_request* = 1.

OFB_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* > 5) 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 values.

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

   Comment: Check the rest of the input parameters.

3. If (*requested_no_of_bits* > 1000), 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*) > 232), then **Return** ("*Additional_input* too long", *Null*).

6. If ((*prediction_resistance_request* = 1) and (*prediction_resistance_flag* ≠ 1)), then **Return** ("Invalid *prediction_resistance_request*", Null).

7. *reseed_required_flag* = 0.

8. If ((*reseed_required_flag* = 1) or (*prediction_resistance_request* = 1)), then do

   Comment: Reseed.

   8.1 *status* = **Reseed_OFB_DRBG_Instantiation** (*state_handle*, *additional_input*).

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

   8.3 *V* = *internal_state* (*state_handle*).*V*, *Key* = *internal_state*

(*state_handle*).*Key*, *reseed_counter* = *internal_state* (*state_handle*).*reseed_counter*.

8.4 *additional_input* = *Null*.

8.5 *reseed_required_flag* = 0.

9. (*status*, *pseudorandom_bits*, *V*, *Key*, *reseed_counter*) = **Generate_algorithm** (*V*, *Key*, *reseed_counter*, *requested_number_of_bits*, *additional_input*).

10. If (*status* ≠ "Success"), then

10.1 *reseed_required_flag* = 1.

10.2 Go to step 8.

11. *internal_state* (*state_handle*) = {*V*, *Key*, *security_strength*, *reseed_counter*, *prediction_resistance_flag*}.

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

Generate_algorithm (...):

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

integer (*state_handle*, *requested_number_of_bits*).

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

Process:

1. If (*reseed_counter* > *reseed_interval*), then **Return** ("Reseed required").

2. If (*additional_input* ≠ *Null*), then

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*).

3. *temp* = *Null*.

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

4.1 *V* = = **TDEA_ECB_Encrypt** (*Key*, *V*).

4.2 *temp* = *temp* || *V*.

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

6. *zeros* = 0^{232}. Comment: Produce a string of *seedlen* zeros.

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

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

   Comment: Save the new values of *V*, *Key* and *reseed_counter*.

9. **Return** ("Success", *returned_bits, V, Key, reseed_counter*).

F.6 Dual_EC_DRBG Example

F.6.1 Discussion

This example of **Dual_EC_DRBG** allows a consuming application to instantiate using any of the recommended elliptic curves, depending on the security strength. A reseed capability is available, but prediction resistance is not available. Both a *personalization_string* and *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 intantiation (*instantiation_nonce*) consists of a random value with 64-bits of entropy; the nonce is obtained by a separate call to the **Get_entropy** routine.

The internal state contains values for *s*, *curve_type*, *seedlen*, *p*, *a*, *b*, *n*, *P*, *Q*, *block_counter* and *security_strength*. In accordance with Table 5 in Section 10.3.2.1, security strengths of 112, 128, 192 and 256 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.
3. Required minimum entropy for instantiation and reseed = *security_strength.*
4. Minimum entropy input length (*min _length*): See Table.
5. Maximum entropy input length (*max _length*) = 1000.
6. Maximum personalization string length (*max_personalization_string_length*) = 500.
7. Maximum additional input length (*max_additional_input_length*) = 500.
8. Seed length (*seedlen*): See Table.
9. Maximum number of bits per request (*max_number_of_bits_per_request*) = 1000.
10. Reseed interval (*reseed_interval*) = 10,000.

F.6.2 Instantiation of Dual_EC_DRBG

This implementation will return a test message and an invalid state handle (-1) when an **ERROR** is encountered. A DRBG-specific parameter *requested_curve_type* is required (rather than optional) for this implementation for a consuming application to select a curve type. **Hash_df** is specified in Section 9.6.2.

Instantiate_Dual_EC_DRBG (...):

Input: integer (*requested_instantiation_security_strength*), bitstring *personalization_string*, integer *requested_curve_type*.

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*) > 500), then **Return** ("*personalization_string* too long", -1).
3. If ((*requested_curve_type* ≠ *Prime_field_curve*) and (*requested_curve_type* ≠ *Random_binary_curve*) and (*requested_curve_type* ≠ *Koblitz_curve*)), then Return ("Valid curve type not specified", -1).

Comment : Determine an *m* that is appropriate for the *requested_strength*; this will depend on *curve_type*.

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

Comment : Choose one of the prime field curves

4.1 If (*requested_instantiation_security_strength* ≤ 112), then

{*security_strength* = 112; *seedlen* = 224; *outlen* = 208; *min_entropy_input_len* = 224}

Else if (*requested_instantiation_security_strength* ≤ 128), then

{*security_strength* = 128; *seedlen* = 256; *outlen* = 240; *min_entropy_input_len* = 256}

Else if (*requested_instantiation_security_strength* ≤ 192), then

{*security_strength* = 192;, *seedlen* = 384; *outlen* = 368; *min_entropy_input_len* = 384}

Else {*security_strength* = 256;, *seedlen* = 521; *outlen* = 504; *min_entropy_input_len* = 528}.

4.2 Select elliptic curve P-*seedlen*, if available. If this curve is not available, then **Return** ("*Prime_field_curve* of the correct length not available", -1).

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

Comment: choose one of the binary or Koblitz curves.

5.1 If (*requested_ instantiation_security_strength* $\leq$ 112), then

{*security_strength* = 112; *seedlen* = 233; *outlen* = 216; *min_entropy_input_len* = 240}

Else if (*requested_ instantiation_security_strength* $\leq$ 128), then

{*security_strength* = 128; *seedlen* = 283; *outlen* = 264; *min_entropy_input_len* = 288}

Else if (*requested_ instantiation_security_strength* $\leq$ 192), then

{*security_strength* = 192; *seedlen* = 409; *outlen* = 392; *min_enropy_input_length* = 416}

Else {*security_strength* = 256; *seedlen* = 571; *outlen* = 552; *min_enropy_input_length* = 576}

5.2 *p* = 0.

5.3 If (*curve_type* = *Random binary_curve*), then select elliptic curve B-*seedlen*; if this curve is not available, then **Return** ("*Random_binary_curve* of the correct length not available", -1).

Else select elliptic curve K-*seedlen*; if this curve is not available, then **Return** ("*Koblitz_curve* of the correct length not available", -1).

6 Set the point *P* to the generator *G* for the curve, and set *n* to the order of *G*.

7. Set the corresponding point *Q* from Annex A.1.

Comment: Request *entropy_input*.

8. (*status*, *entropy_input*) = **Get_entropy** (*security_strength*, *min _length*, 1000).

9. If (*status* $\neq$ "Success"), then **Return** ("Failure indication returned by the *entropy_input* source:" || *status*, -1).

10. (*status*, *instantiation_nonce*) = **Get_entropy** (64, 64, 1000).

11. If (*status* $\neq$ "Success"), then **Return** ("Failure indication returned by the random nonce source:" || *status*, -1).

Comment : Perform the instantiate algorithm.

12. *seed_material* = *entropy_input* || *instantiation_nonce* || *personalization_string*.

13. *s* = **Hash_df** (*seed_material*, *seedlen*).

14. *block_counter* = 0.

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

15. (*status*, *state_handle*) = **Find_state_space** ( ).

16. If (*status* ≠ "Success"), then **Return** (*status*, -1).

17. *internal_state* (*state_handle*) = {*s*, *curve_type*, *m*, *p*, *a*, *b*, *n*, *P*, *Q*, *block_counter*, *security_strength*}.

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

F.6.3 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_string*.

Output: string *status*.

Process:

Comment: Check the input parameters.

1. If ((*state_handle* > 10) or (*internal_state* (*state_handle*).*security_strength* = 0)), then **Return** ("State not available for the *state_handle*").

2. If (**len** (*additional_input*) > 500), 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** (*security_strength*, *min_ entropy_input_length*, 1000).

4. If (*status* ≠ "Success"), then **Return** ("Failure indication returned by 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*).

7. *block_counter* = 0.

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

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

F.6.4 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* > 10) 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*, *security_strength* = *internal_state* (*state_handle*).*security_strength*, *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) > 500), then **Return** ("Additional_input too long", *Null*).

Comment: Check whether a reseed is required.

6. If $\left(block\_counter + \left\lceil \frac{requested\_number\_of\_bits}{outlen} \right\rceil\right) > 10{,}000$, then

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

   6.2 *additional_input* = Null.

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

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 11.
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*).

F.7 MS_DRBG Example

F.7.1 Discussion

This example of **MS_DRBG** allows a consuming application to request specific values for *e* and *outlen*. A reseed capability is available, but prediction resistance is dependent on the user's system. Both a *personalization_string* and *additional_input* are allowed. A total of 5 internal states are provided. For this implementation, the handling of the DRBG-specific parameters and the algorithms are provided as separate routines.

The nonce for intantiation consists of a random value with 64-bits of entropy; the nonce is obtained by increasing the call for entropy bits via the **Get_entropy** call by 64 bits (i.e., by adding 64 bits to the *security_strength* value).

The internal state contains values for *n*, *e*, *seedlen*, *outlen*, *S*, *block_counter*, *security_strength* and *prediction_resistance_flag*.

In accordance with Table 6 in Section 10.3.3.1, security strengths of 112 and 128 may be supported. SHA-1 has been selected as the hash function. The following definitions are applicable for the instantiate, reseed and generate functions :

1. *highest_supported_security_strength*: Depends on the requested security strength.
2. Output block length (*outlen*): 8, unless otherwise requested using *requested_outlen*.
3. Required minimum entropy for instantiation = *security_strength* + 64 (includes the randm nonce).
4. Required minimum entropy for reseed = *security_strength*.
5. Minimum entropy input length (*min _length*): *min_entropy*.
6. Maximum entropy input length (*max _length*) = 5000 bits.
7. Maximum personalization string length (*max_personalization_string_length*) = 500 bits.
8. Maximum additional input length (*max_additional_input_length*) = 500 bits.
9. Number of hard bits = 11.
10. Seed length (*seedlen*): lg (n) - 8.
11. Maximum number of bits per request (*max_number_of_bits_per_request*) = 200,000 bits.
12. Reseed interval (*reseed_interval*) = 25,000 blocks of *outlen* bits.

F.7.2 Instantiation of MS_DRBG

This implementation will return a test message and an invalid state handle (-1) when an **ERROR** is encountered. DRBG-specific parameters (*requested_e* and *requested_outlen*) are provided that will allow a consuming application to optionally select the values for *e* and *outlen*. **Hash_df** is specified in Section 9.6.2.

If *prediction_resistance_flag* = 1, then a prediction resistance capability is requested for the instantiation. If the user's system is capable of handling prediction resistance (e.g., a source of randomness is readily available), the user has been instructed to indicate the ability to provide prediction resistance by setting *prediction_resistance_capability* = 1 during system configuration.

Let **Get_random_modulus** be a function that gets a random modulus *n* that meets the criteria specified in Section 10.3.3.2.3, step 5.5.

Instantiate_MS_DRBG (...):

Input: integer (*requested_instantiation_security_strength*, *prediction_resistance_flag*), bitstring *personalization_string*, integer (*requested_e* , *requested_outlen*).

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

Process:

1. If (*requested_ instantiation_security_strength* > 128), then **Return** ("Invalid *requested_ instantiation_security_strength*", -1).

2. If ((*prediction_resistance_flag* = 1) and (*prediction_resistance_capability* ≠ 1)), then **Return** ("Cannot support prediction resistance", -1).

3. If (**len** (*personalization_string*) > 500), then **Return** ("Personalization_string too long", -1).

4. If (*requested_ instantiation_security_strength* ≤ 112), then *security_strength* = 112

   Else *security_strength* = 128.

5. (*status*, *n*, *e*, *seedlen*, *outlen*) = **Get_DRBG_specific_parameters** (*security_strength*, *requested_e*, *requested_outlen*).

   Comment: Get *entropy_input*.

6. *min_entropy* = *security_strength* + 64.

7. (*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_entropy*, 5000).

8. If ($status \neq$ "Success"), then **Return** ("Failure indication returned by the entropy source", -1).

9. (S, *block_counter*) = **Instantiate_algorithm** (*entropy_input, personalization_string, seedlen*).

Comment: Find an empty state in the state space.

10. (*status, state_handle*) = **Find_state_space** ( ).

11. If ($status \neq$ "Success "), **Return** (*status*, -1).

Comment: Store all values in *state* .

12. *internal_state* (*state_handle*) = {*n, e, seedlen, outlen, S, block_counter, security_strength, prediction_resistance_flag*}.

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

Get_DRBG_specific_parameters (...).

Input: integer (*security_strength, requested_e, requested_outlen*).

Output: string (*status*), integer (*n, e, seedlen, outlen*).

Process:

Comment: Determine modulus size (i.e., lg(n)).

1. If (*security_strength* = 112) then *modulus_size* = 2048

   Else *modulus_size* = 3072.

Comment: Select the exponent e.

2. If (*requested_e* = 0) or is not provided, then $e = 3$

   Else

   2.1 e = *requested_e*.

   2.2 If (($e < 3$) or ($e > (2^{\lg(n)-1} - (2 \times 2^{1/2 \lg(n)}))$) or ($e \bmod 2 = 0$)), then **Return** ("Invalid *requested_e*", -1).

Comment: Determine *outlen*.

3. If (*requested_outlen* = 0) or is not provided, then *outlen* = 8

   Else

   3.1 *outlen* = *requested_outlen*.

   3.2 If ((*outlen* < 1) or (*outlen* > **min** ($\lfloor \lg(n) - 2 * security\_strength \rfloor$, $\lfloor \lg(n) * (1 - 2/e) \rfloor$) or (*outlen* mod 8 $\neq$ 0)), then **Return** ("Inappropriate value for *requested_outlen*", -1).

4. *seedlen* = *modulus_size* – *outlen*. Comment: Determine the seed length.

   Comment: Select the modulus *n*.

5. (*status*, *n*) = **Get_random_modulus** (*modulus_size*, *e*).

6. If (*status* ≠ "Success "), then **Return** ("Failed to produce an appropriate modulus", -1).

7. **Return** ("Success", *n*, *e*, *seedlen*, *outlen*).

Instantiate_algorithm (...):

Input: bitstring (*entropy_input*, *personalization_string*), integer *seedlen*.

Output: integer (*S*, *block_counter*).

Process:

1. *seed_material* = *entropy_input* || *personalization_string*.
2. *S* = **Hash_df** (*seed_material*, *seedlen*).
3. *block_counter* = 0.
4. **Return** (*S*, *block_counter*).

F.7.3 Reseeding an MSDRBG Instantiation

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

Reseed_MS_DRBG (...):

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

Output: string *status*.

Process:

1. If ((*state_handle* > 5) or (*internal_state* (*state_handle*).*security_strength* = 0)), then **Return** ("State not available for the indicated *state_handle* ").

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

2. *S* = *internal_state*(*state_handle*).*S*, *seedlen* = *internal_state*(*state_handle*).*seedlen*, *security_strength* = *internal_state* (*state_handle*).*security_strength*.

3. If (**len** (*additional_input*) > 500), then **Return** ("*Additional_input* too long ", -1).

4. *min_entropy* = *security_strength*.

5. (*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_entropy*, 5000).

6. If ($status \neq$ "Success"), then **Return** ("Failure indication returned by the *entropy_input* source ").
7. (S, *block_counter*) = **Reseed_algorithm** (*entropy_input*, *additional_input*, S, *seedlen*).
8. *internal_state* (*state_handle*).S = S, *internal_state* (*state_handle*), *block_counter* = *block_counter*.
9. **Return** ("Success").

Reseed_algorithm (...):

Input: bitstring (*entropy_input*, *additional_input*), integer (S, *seedlen*).

Output: integer (S, *block_counter*).

Process:

1. *seed_material* = S || *entropy_input* || *additional_input*.
2. S = **Hash_df** (*seed_material*, *seedlen*).
3. *block_counter* = 0.
4. **Return** (S, *block_counter*).

F.7.4 Generating Pseudorandom Bits Using MS_DRBG

The implementation returns a *Null* string as the pseudorandom bits if an error is encountered. If prediction resistance is needed, then *prediction_resistance_request* = 1.

MS_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:

1. If ((*state_handle* > 5) or (*internal_state* (*state_handle*).*security_strength* = 0)), then **Return** ("State not available for the indicated *state_handle* ", *Null*).

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

2. S = *internal_state* (*state_handle*).S, n = *internal_state* (*state_handle*).n, e = *internal_state* (*state_handle*).e, *outlen* = = *internal_state* (*state_handle*).*outlen*, *seedlen* = *internal_state* (*state_handle*).*seedlen*, *security_strength* = *internal_state* (*state_handle*).*security_strength*, *block_counter* = *internal_state* (*state_handle*).*block_counter*, *prediction_resistance_flag* = *internal_state* (*state_handle*). *prediction_resistance_flag*.
3. If (*requested_no_of_bits* > (25000 × *outlen*)), 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*) > 500), then **Return** ("Additional_input too long", *Null*).
6. If ((*prediction_resistance_request* = 1) and (*prediction_resistance_flag* ≠ 1)), then **Return** ("Prediction resistance capability not instantiated", *Null*).
7. *reseed_required_flag* = 0.
8. If ((*reseed_required_flag* = 1) or (*prediction_resistance_request* = 1)), then
   8.1 *status* = **Reseed_MS_DRBG** (*state_handle*, *additional_input*).
   8.2 *S* = *internal_state* (*state_handle*).*S*, *block_counter* = *internal_state* (*state_handle*).*block_counter*.
   8.3 *additional_input* = *Null*.
   8.4 *reseed_request_flag* = 0.
9. (*status*, *pseudorandom_bits*, *S*, *block_counter*) = **Generate_algorithm** (*n*, *e*, *seedlen*, *outlen*, *S*, *block_counter*, *requested_number_of_bits*, *additional_input*).
10. If (status ≠ "Success"), then
    10.1 *reseed_required_flag* = 1.
    10.2 Go to step 8.
11. *internal_state*.*S* = *S*, *internal_state*.*block_counter* = *block_counter*.
12. **Return** ("Success", *pseudorandom_bits*).

Generate_algorithm (...):

Input: integer (*n*, *e*, *seedlen*, *outlen*, *S*, *block_counter*, *requested_number_of_bits*), bitstring *additional_input*.

Output: string status, bitstring *pseudorandom_bits*.

Process:

1. If $\left(\left(reseed\_counter + \left\lceil \frac{requested\_number\_of\_bits}{outlen} \right\rceil\right) > 25{,}000\right)$, then Return ("Reseed required", *Null*).
2. If (*additional_input* = *Null*), then *additional_input* = 0
   Else *additional_input* = **Hash_df** (**pad8** (*additional_input*), *seedlen*).

3. *temp* = the *Null* string.
4. $i = 0$.
5. $s = S \oplus additional\_input$.
6. $S = [\,(s^e \bmod n) / 2^{seedlen}\,]$. Comment: S is an *seedlen*-bit number.
7. $R = (s^e \bmod n) \bmod 2^{outlen}$. Comment: R is an *outlen*-bit number.
8. $temp = temp \,||\, R$.
9. $additional\_input = 0^{seedlen}$.
10. $i = i + 1$.
11. $block\_counter = block\_counter + 1$.
12. If (**len** (*temp*) < *requested_no_of_bits*), then go to step 6.
13. *pseudorandom_bits* = **Truncate** (*temp*, $i \times outlen$, *requested_no_of_bits*).
14. **Return** ("Success", *pseudorandom_bits*).

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 140-2, *Security Requirements for Cryptographic Modules*, May 25, 2001.

[9] National Institute of Standards and Technology Special Publication 800-38A, *Recommendation for Block Cipher Modes of Operation - Methods and Techniques*, December 2001.

[10] NIST Special Publication 800-57 (Draft), *Recommendation for Key Management*, [Insert date].