DRAFT X9.82 (Random Number Generation) Part 3, Deterministic Random Bit Generator Mechanisms Late August 2005

Table of Contents

Random Number Generation

Part 3: Deterministic Random Bit Generator Mechanisms

Contribution of the U.S. Federal Government and not subject to copyright

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

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

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

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

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

FIPS 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 $0 \le r \le n\text{-}1$) 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.
$\{a_1, \dots a_i\}$	The internal state of the DRBG at a point in time. The types and number of the a_i depends on the specific DRBG.
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: 12
> 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 provides a discussion on DRBG selection.

— Annex E provides example pseudocode for each DRBG.

— Annex F 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.
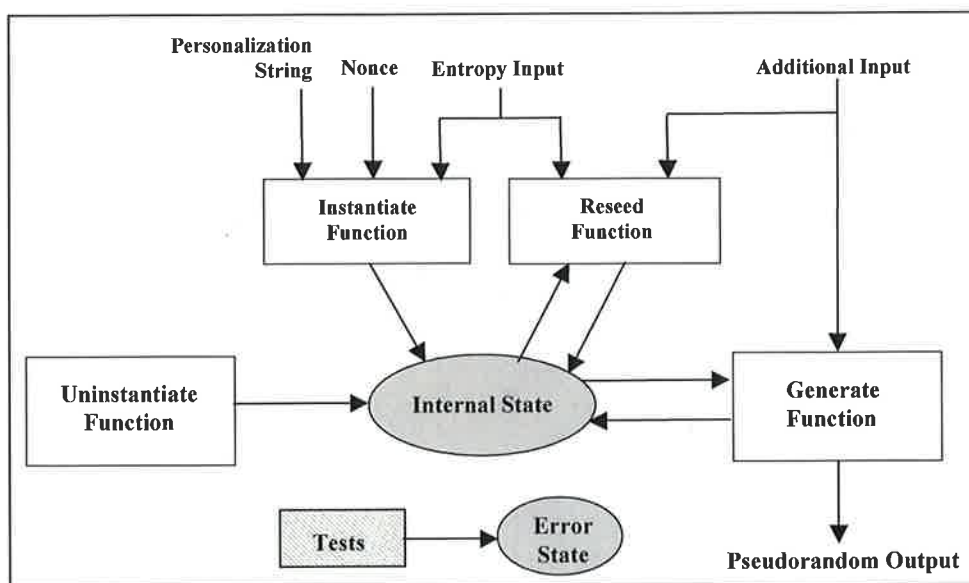
Figure 1: DRBG Functional Model

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: 14
> 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 entropy 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 8.4.2.

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

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

7.2.4   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

15

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

Figure 2: DRBG Instantiation

   b. A count of the number of requests or blocks produced since the instantiation

17

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

19

DRBG does not require repeated use of the instantiate or reseed functions.
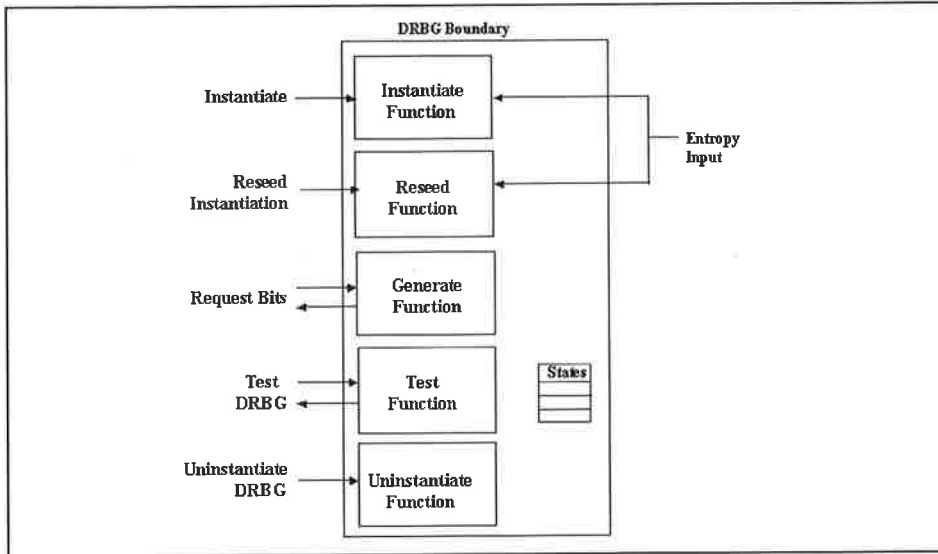
Figure 3: DRBG Functions Within a Single Device

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.
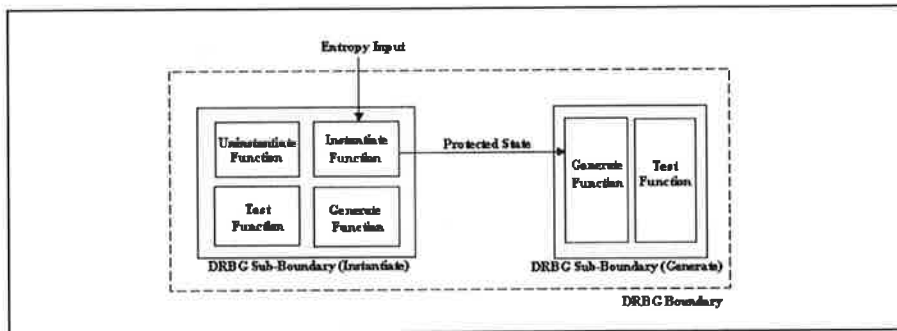
Figure 4: Distributed DRBG Functions

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.
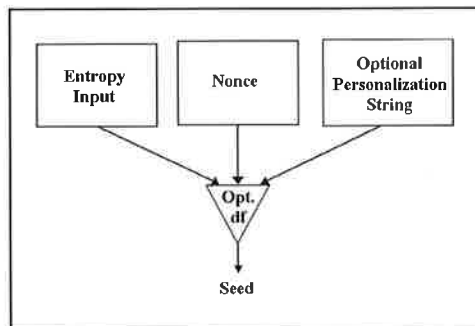
   Figure 5: Seed Construction for Instantiation

   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

21

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.
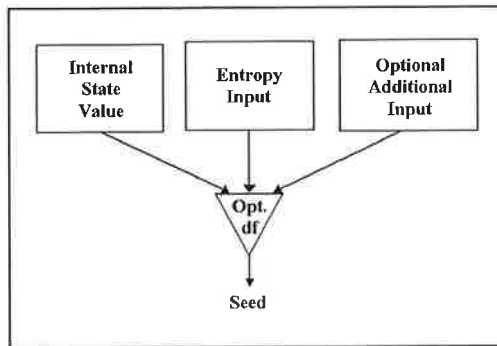
Figure 6: Seed Construction for Reseeding

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.

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 Part 1). 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, 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 current internal state 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.

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 8.4.2). 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.
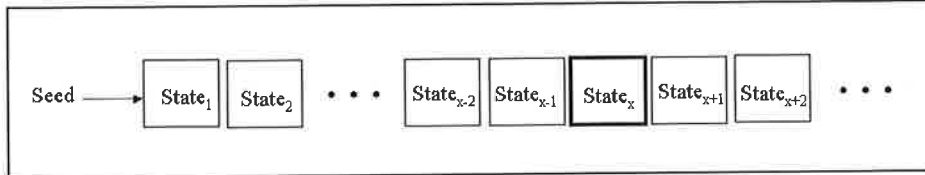
Figure 7: Sequence of DRBG States

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 $State_{x-1}$. 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

25

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 [ebb3]:** Page: 25
> 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 adversaryhe 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**
> **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 $State_{x-2}$.

$State_{x-1}$ 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 currentprevious DRBG internal state does not allow an adversary any useful knowledge about future DRBG internal states or outputs.

26

9   DRBG Functions

9.1   General Discussion

The DRBG functions in this Standard are specified as an algorithm (see Section 10) and an "envelope" of pseudocode around that algorithm (defined in this section). The pseudocode in the envelopes checks the input parameters, obtains input not provided by the input parameters, accesses the appropriate DRBG algorithm and handles the internal state. A function need not be implemented using such envelopes, 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 (see SP 800-67); for AES, the basic encryption operation is called the cipher operation (see FIPS 197). 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:

1. Checks the validity of the other input parameters,

2. Determines the security strength for the DRBG instantiation,

3. Determines any DRBG specific parameters (e.g., elliptic curve domain parameters),

4. Obtains entropy input with entropy sufficient to support the security strength,

27

5. Obtains the nonce,

6. Determines the initial internal state using the instantiate algorithm,

7. Returns 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). 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 11.

3. *personalization_string*: An optional input that provides personalization information (see Sections 8.4.2 and 8.5.2). The maximum length of the personalization string (*max_personalization_string_length*) is implementation dependent, but **shall** be ≤ the maximum length specified for the given DRBG (see Section 10). If a personalization string will never be used, then the input parameter and step 3 may be omitted, and step 9 may be modified to omit the personalization string.

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 ≤ 2^{35} bits.

2. *nonce*: A nonce as specified in Section 8.4.2. Note that if a random value is used as the nonce, the *entropy_input* and *nonce* could be acquired using a single **Get_entropy** 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.

28

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 return 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). Otherwise, the step **should** be omitted.

5. Using the *security_strength*, select appropriate DRBG parameters.

> Comment: Obtain the entropy input.

6. (*status, entropy_input*) = **Get_entropy** (*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 8.4.2.

> Comment: Call the appropriate instantiate algorithm in Section 10 to obtain values for

29

the initial *working_state*.

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

10. Get a *state_handle* for a currently empty state. If an unused internal state cannot be found, return an **ERROR**.

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

12. 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,

- performed when prediction resistance is requested by an application,

- triggered by the generate function after a predetermined number of pseudorandom outputs have been produced or a pre-determined number of requests have been made, or

- triggered by external events (e.g., whenever sufficient entropy is available).

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

The reseed function:

1. Checks the validity of the input parameters,

2. Obtains entropy input with sufficient entropy to support the security strength, and

3. Using the reseed algorithm, combines the current internal state with the new entropy input and any additional input to determine the new internal state.

Let *working_state* be the working state for the particular DRBG, and let *min_length* and *max_length* be defined for each DRBG (see Section 10).

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

Input from a consuming application:

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 ≤ the

maximum value specified for the given DRBG (see Section 10). If *additional_input* will never be used, then the input parameter and step 2 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

31

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.

> **Comment [EBB4]:** Page: 32
> Is it really useful to check the successive states during reseeding ? The HMAC and CTR DRBGs have this feature ; the Dual_EC does not (it would be best to make similar checks in all DRBGs.

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:

1.   Checks the validity of the input parameters,

2.   Calls the reseed function to obtain sufficient entropy if the instantiation needs additional entropy because the end of the seedlife has been reached or prediction resistance is required.

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

4.   Updates the working state.

5.   Returns the requested pseudorandom bits to the consuming application.

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 prediction resistance is never provided, then the *prediction_resistance_request*

32

input parameter and 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 the *prediction_resistance_request* input parameter and step 5 may be omitted, and steps 7 and 8 are replaced by:

> *status* = **Reseed** (*state_handle, additional_input*).
>
> If *status* indicates an **ERROR**, then return **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

33

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: Check whether a reseed is needed.

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.

11. Replace the old *working_state* in the internal state indicated by *state_handle* with the new *working_state*.

12. 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". The uninstantiate function:

1. Checks the input parameter for validity.

2. Empties the internal state.

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

Input from a consuming application:

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 Section 9.6.3). The block cipher derivation function uses the **Block_Cipher_Hash** function 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 ≤ (255 × *outlen*). The *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. $len = \left\lceil \dfrac{no\_of\_bits\_to\_return}{outlen} \right\rceil.$

4. *counter* = an 8-bit binary value representing the integer "1".

5. For *i* = 1 to *len* do

    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 (Block_Cipher_df)

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:

36

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 representation 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. $S = L \parallel N \parallel$ *input_string* $\parallel$ 0x80.

   Comment : Pad S with zeros, if necessary.

4. While (**len** (S) mod *outlen*) $\neq$ 0, $S = S \parallel$ 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 $IV = i \parallel 0^{outlen - \mathbf{len}\ (i)}$.

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

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

   8.3 $i = i + 1$.

37

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 operation.

2. *data_to_hash*: The data to be operated upon. Note that the length of *data_to_hash* must be a multiple of *outlen*. This is guaranteed by steps 4 and 8.1 in Section 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

Known-answer tests on the instantiate function **shall** be performed prior to creating each operational instantiation. However, if several instantiations are performed in quick succession using the same input parameters, then the testing **may** be reduced to testing prior to creating the first instantiation using that parameter set.

The *security_strength* and *prediction_resistance_flag* to be used in the operational invocation **shall** be used during the test. Representative fixed values and lengths of the *entropy_input*, *nonce* and *personalization_string* (if allowed) **shall** be used; the value of the *entropy_input* used during testing **shall not** be intentionally reused during normal operations (either by the instantiate or the reseed functions). Error handling **shall** 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* and *prediction_resistance_flag*.

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

9.7.3 Testing the Generate Function

Known-answer tests **shall** be performed on the generate function before the first use of the function and at reasonable intervals defined by the implementer. The implementer **shall** document the intervals and provide a justification for the selected intervals.

The known-answer tests **shall** be performed for each implemented *security_strength*. Representative fixed values and lengths for the *requested_number_of_bits* and *additional_input* (if allowed) and the working state of the internal state value (see Sections 8.2.3 and 10) **shall** be used. If prediction resistance is available, then each combination of the *security_strength*, *prediction_resistance_request* and *prediction_resistance_flag* **shall** be tested. The error handling for each input parameter **shall** also be tested, and testing **shall** include setting the *reseed_counter* to meet or exceed the *reseed_interval* in order to check that the implementation is reseeded or that the DRBG is "shut down", as appropriate.

If the values used during the test produce the expected results, and errors are handled

39

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.

Note that the generate function performs a continuous test by comparing sequential output blocks; for some DRBGs, additional values of the working state are checked to determine that they have changed (see the generate function for each DRBG in Section 10 and step 10 in Section 9.4).

Comment [EBB5]: Page: 40
Do we want to do this ?

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 **shall** be used (see Sections 8.2.3 and 10). Error handling **shall** also be tested, including an error in obtaining the *entropy_input* (e.g., the *entropy_input* source is broken).

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

The reseed function may be called every time that the generate function is called if prediction resistance is available, and considerbly less frequently otherwise. Self-test shall be performed as follows:

1. When prediction resistance is available in an implementation, the reseed function **shall** be tested whenever the generate function is tested (see above).

2. When prediction resistance is not available in an implementation, the reseed function **shall** be tested whenever the reseed function is invoked and before the reseed is performed on the operational instantiation.

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

Note that for some DRBGs, the reseed function performs a continuous test by comparing working state values after reseeding with the working state values before reseeding (see the reseed function for each DRBG in Section 10 and step 6 in Section 9.3).

Comment [EBB6]: Page: 40
Do we want to do this ?

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".

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.

41

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 D. Pseudocode examples for each DRBG are provided in Annex E. Conversion specifications required for the DRBG implementations (e.g., between integers and bitstrings) are provided in Annex B.

10.1 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. This Standard supports only four security strengths for DRBGs: 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	440	440	440	888	888
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.

10.1.3 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 by 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.

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

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

2. Administrative information:
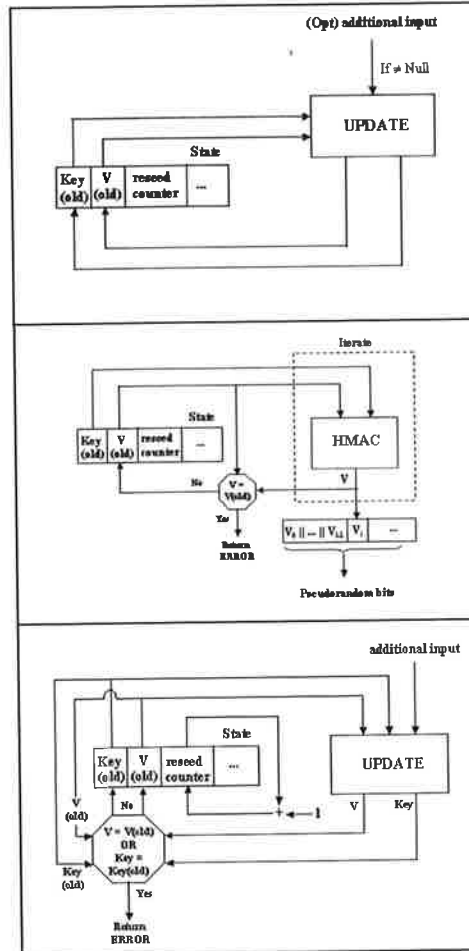
Figure 9: HMAC_DRBG Generate Function

a. The *security_strength* of the DRBG instantiation.

b. A *prediction_resistance_flag* that indicates whether or not a prediction resistance capability is required for the DRBG instantiation.

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

10.1.3.2.2 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.

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

Figure 10: HMAC_DRBG 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, 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 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 8.4.2.

3. *personalization_string*: The personalization string received from the consuming application. If a *personalization_string* will never be used, then step 1 may be modified to remove the *personalization_string*.

Output:

1. *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* = 0x00 00...00.   Comment: *outlen* bits.

3. V = 0x01 01...01.   Comment: *outlen* bits.

                                      Comment: Update *Key* and V.

4. (*Key*, V) = **Update** (*seed_material*, *Key*, V).

                                      Comment: Generate the initial block for comparing with the 1st DRBG output block (for continuous testing)

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

6. (*Key*, V) = **Update** (*seed_material*, *Key*, V).

5. *reseed_counter* = 1.

6. Return *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 be modified 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 that the old and new values of *Key* and *V* are not identical.

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*, *Key* 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 3 may be omitted. If an implementation does not include the *additional_input* parameter as one of the calling parameters, or if the implementation allows *additional_input*, but a given request does not provide any *additional_input*, then a *Null* string **shall** be used as the *additional_input* in step 7.

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. *Key_old* = *Key*; *V_old* = *V*.

3. If *additional_input* ≠ *Null*, then

    3.1   (*Key*, *V*) = **Update** (*additional_input*, *Key_old*, *V_old*).

48

> Comment: Continuous test - Check for that successive values of *V* and *Key* are not identical.

   3.2   If (($Key = Key\_old$) OR ($V = V\_old$)), then return and ERROR.

   3.3   *Key_old* = *Key*; *V_old* = *V*.

                                                                                      Comment [EBB7]: Page: 49
                                                                                    Is this additional test useful ?

4.   *temp* = *Null*.

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

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

> Comment: Continuous test - Check for that successive values of *V* are not identical.

   5.2   If ($V = V\_old$), then return an **ERROR**.

   5.3   *V_old* = *V*.

   5.4   *temp* = *temp* ‖ *V*.

6.   *returned_bits* = Leftmost *requested_number_of_bits* of *temp*.

7.   (Key, V) = **Update** (*additional_input, Key_old, V_old*).

> Comment: Continuous test - Check for that successive values of *V* and *Key* are not identical.

8.   If (($V = V\_old$) or ($Key = Key\_old$)), then return an **ERROR**.

                                                                                    Comment [EBB8]: Page: 49
                                                                                    Is this additional test useful ?

9.   *reseed_counter* = *reseed_counter* + 1.

10. Return **SUCCESS**, *returned_bits*, and the new values of *Key*, *V* and *reseed_counter* as the *working_state*).

10.2 DRBG Based on Block Ciphers

10.2.1 Discussion

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

10.2.2 CTR_DRBG

10.2.2.1 CTR_DRBG Description

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.

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.

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

Figure 11: CTR_DRBG Update Function

Table 4: Definitions for the CTR_DRBG

	3 Key TDEA	**AES-128**	**AES-192**	**AES-256**

	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			
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 CTR_ DRBG may be implemented to use the block cipher derivation function

specified in Section 9.6.3. However, the DRBG is 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 **shall not** use a nonce[2] (see Section 8.4.2).

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

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 10.2.2.1).
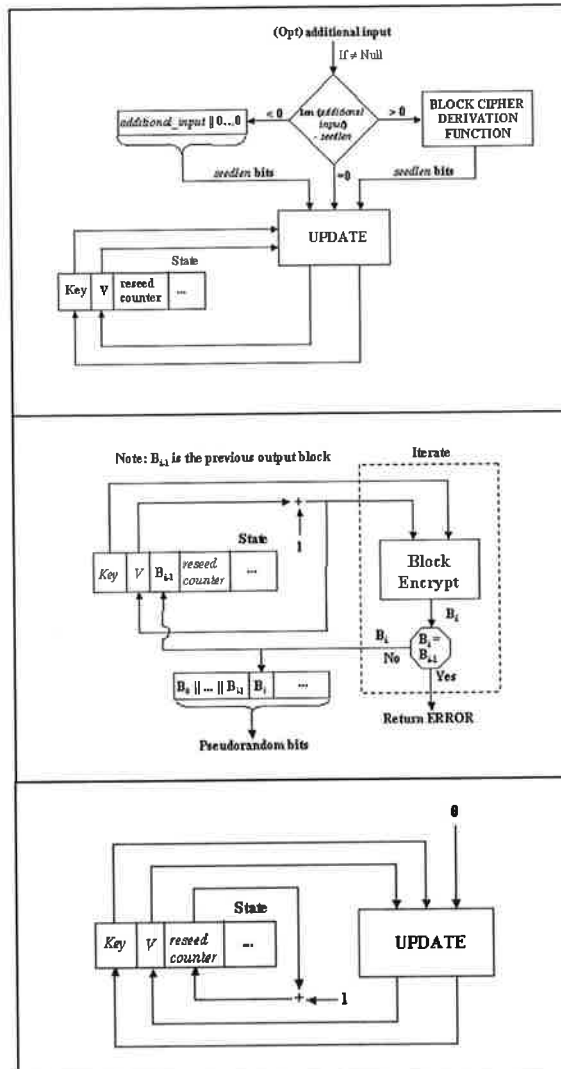
Figure 12: CTR-DRBG

---

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

b. The *keylen*-bit *Key*, which is updated whenever a predetermined number of output blocks are generated.

c. The *previous_ output_block*; this is required to perform a continuous test on the output from the generate function.

d. 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*, *Key* and *previous_ output_block* are the critical values of the internal state upon which the security of this DRBG depends (i.e., *V*, *Key* and *previous_ output_block* 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 10.2.2.1. 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.*

3. *temp* = Leftmost *seedlen* bits of *temp.*

4   *temp = temp* $\oplus$ *provided_data.*

5. *Key* = Leftmost *keylen* bits of *temp.*

6. *V* = Rightmost *outlen* bits of *temp.*

7. Return the new values of *Key* and V.

10.2.2.2.3 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, step 5 **should** be omitted. The values of *highest_supported_security_strength* and *min _length* are provided in Table 4 of Section 10.2.2.1. The contents of the internal state are provided in Section 10.2.2.2.1.

The instantiate algorithm:

Let **Update** be the function specified in Section 10.2.2.2.2. The output block length (*outlen*), key length (*keylen*), seed length (*seedlen*) and *security_strengths* for the block cipher algorithms are provided in Table 4 of Section 10.2.2.1.

If a block cipher derivation function is to be used, then the **Block_Cipher_df** specified in Section 9.6.3 **shall** be implemented using the chosen block cipher algorithm and key size; in this case, step 1 below **shall** consist of steps 1.1 and 1.2 (i.e., steps 1.3 to 1.5 **shall not** be used).

If full entropy is available whenever entropy input is required, and a block cipher derivation function is not to be used, then step 1 below **shall** consist of steps 1.3 to 1.5 (i.e., steps 1.1 and 1.2 **shall not** be used).

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 8.4.2; this string **shall not** be present unless a derivation function is used.

3. *personalization_string*: The personalization string received from the consuming application.

Output:

1. *working_state*: The inital values for *V, Key, previous_output_block* 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: If 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 *personalization_string = personalization_string* ‖ $0^{seedlen - temp}$.

    1.5   *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.

                                   Comment: Generate the initial block for comparing with the 1st DRBG output block (for continuous testing)

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

7. *zeros* = $0^{seedlen}$.   Comment: Produce a string of *seedlen* zeros.

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

9. Return *V, Key, first_output_block* and *reseed_counter* as the *working_state.*

<u>Implementation notes:</u>

1. 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*).

2. 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 – 1.5 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 10.2.2.1.

The reseed algorithm:

> Let **Update** be the function specified in Section 10.2.2.2.2. The seed length (*seedlen*) is provided in Table 4 of Section 10.2.2.1.

> If a block cipher derivation function is to be used, then the **Block_Cipher_df** specified in Section 9.6.3 **shall** be implemented using the chosen block cipher algorithm and key size; in this case, step 1 below **shall** consist of steps 1.1 and 1.2 (i.e., steps 1.3 to 1.5 **shall not** be used).

> If full entropy is available whenever entropy input is required, and a block cipher derivation function is not to be used, then step 1 below **shall** consist of steps 1.3 to 1.5 (i.e., steps 1.1 tand 1.2 **shall not** be used).

> 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, previous_output_block* 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, previous_output_block* 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 *additional_input* = *additional_input* ∥ $0^{seedlen-temp}$.

    1.5   *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 SUCCESS,** *V*, *Key*, *previous_output_block* and *reseed_counter* as the *working_state*.

Implementation notes:

1.   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*).

2.   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 − 1.5 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 10.2.2.1. If the derivation function is not used, then the maximum allowed length of *additional_input* = *seedlen*.

Let **Update** be the function specified in Section 10.2.2.2.2. The seed length (*seedlen*) and the value of *reseed_interval* are provided in Table 4 of Section 10.2.2.1. Step 5.2 below uses the selected block cipher algorithm. If a derivation function is not used for a DRBG implementation, then step 3.2 **shall** be omitted.

If a block cipher derivation function is to be used, then the **Block_Cipher_df** specified in Section 9.6.3 **shall** be implemented using the chosen block cipher algorithm and key

size; in this case, step 3.2 below **shall** be included.

If full entropy is available whenever entropy input is required, and a block cipher derivation function is not to be used, then step 2 below **shall not** be used.

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*, *Key*, *previous_output_block* 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.

58

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

3.4 (Key, V) = **Update** ($additional\_input$, Key_old, V_old).

3.5 If (($Key = Key\_old$) OR ($V = V\_old$)), then return an **ERROR.**

> **Comment [EBB9]:** Page: 59
> Is this additional step useful ?

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).

> Comment: Continuous test: Check that the old and new output blocks are different.

5.3 If ($output\_block = previous\_output\_block$), then return an **ERROR**.

5.4 $previous\_output\_block = output\_block$.

5.5 $temp = temp \parallel ouput\_block$.

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. $Key\_old = Key$; $V\_old = V$.

9. (Key, V) = **Update** ($zeros$, Key_old, V_old).

10. If (($V = V\_old$) or ($Key = Key\_old$)), then return an **ERROR.**

> **Comment [EBB10]:** Page: 59
> Is this additional step useful ?

11. $reseed\_counter = reseed\_counter + 1$.

12 Return **SUCCESS** and $returned\_bits$; also return Key, V, $output\_block$ and $reseed\_counter$ as the new $working\_state$.

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.

10.3.2 Dual Elliptic Curve Deterministic RBG (Dual_EC_DRBG)

10.3.2.1 Discussion

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

Dual_EC_DRBG uses a seed that is m bits in length (i.e., *seedlen* = m) to initiate the generation of *outlen*-bit pseudorandom strings by performing scalar multiplications on two points in an elliptic curve group, where the curve is defined over a field approximately 2^m in size. For all the NIST curves given in this Standard for the DRBG, $m \geq 224$. Figure 15 depicts the **Dual_EC_DRBG**.
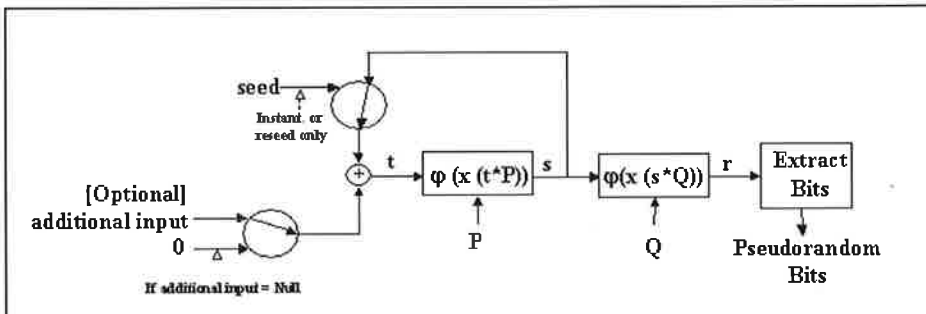
Figure 15: Dual_EC_DRBG

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. Requirements for the *seed* are provided in Section 8.4.2.

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:

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

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

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 (formal definitions for φ and x are given in Section 10.3.2.2.4). Following a line in the direction of the arrow is the normal operation; inverting the direction implies the ability to solve the ECDLP for that specific curve. An adversary's ability to

**Figure 16: Dual_EC_DRBG (...)
Backtracking Resistance**

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	P-256	P-384	P-521
Supported security strengths	See SP 800-57			
highest_supported_ security_strength	See SP 800-57			
Output block length (*max_outlen* = largest multiple of 8 less than *seedlen* - (13 + log₂ (the cofactor))	208	240	368	504
Required minimum entropy for instantiate and reseed	*security_strength*			
Minimum entropy input length $(min\_length = 8 \times \lceil seedlen/8 \rceil)$	224	256	384	528
Maximum entropy input length (*max_length*)	$\leq 2^{13}$ bits			

	P-224	P-256	P-384	P-521
Maximum personalization string length (*max_personalization_string_length*)	$\leq 2^{13}$ bits			
Supported security strengths	See SP 800-57			
Seed length (*seedlen = m*)	224	256	384	521
Appropriate hash functions	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512		SHA-224, SHA-256, SHA-384, SHA-512	SHA-256, SHA-384, SHA-512
max_number_of_bits_per_request	*max_outlen* × *reseed_interval*			
Number of blocks between reseeding (*reseed_interval*)	$\leq 2^{32}$ blocks			

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 (*seedlen, p, a, b, n*), where *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*.

    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. *r_old*, the previous output block.

    e. A counter (*block_counter*) that indicates the number of blocks of random produced by the **Dual_EC_DRBG** since the initial seeding or the previous reseeding.

2. Administrative information:

    a. The *security_strength* provided by the instance of the DRBG,

    b. A *prediction_resistance_flag* that indicates whether prediction resistance is

required by the DRBG.

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

10.3.2.2.2 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.

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 the *security_strength* and Table 5 in Section 10.3.2.1, select the smallest available curve that has a security strength $\geq$ *security_strength*.

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

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

The values for *highest_supported_security_strength* and *min_length* are determined by the selected curve (see Table 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 8.4.2.

3. *personalization_string*: The personalization string received from the consuming application.

Output:

1. s: The initial secret value for the *working_state*.

2. *r_old*: The initial output block (which will not be used).

3. *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: Generate the initial block for comparing with the 1st DRBG output block (for continuous testing).

3. *r_old* = φ(*x*(*s* * *Q*)).   Comment: *r* is a *seedlen*-bit number.

4. *block_counter* = 0.

5. Return *s, r_old* 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

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.

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

3. *block_counter* = 0.

4. Return *s* and *block_counter* for the new *working_state*.

Implementation notes:

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

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

10.3.2.2.4   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 *max_outlen* are provided in Table 5 of Section 10.3.2.1. *outlen* is the number of pseudorandom bits taken from each *x*-coordinate as the **Dual_EC_DRBG** steps. For performance reasons, the value of *outlen* should be set to the maximum value as provided in Table 5. However, an implementation **may** set *outlen* to any multiple of 8 bits less than or equal to *max_outlen*. The bits that become the **Dual_EC_DRBG** output are always the rightmost bits, i.e., the least significant bits of the *x*-coordinates.

The generate algorithm:

Let **Hash_df** be the hash derivation function specified in Section 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, given in affine coordinates. An implementation may choose to represent points internally using other coordinate systems; for instance, when efficiency is a primary concern. In this case, a point **shall** be translated back to affine coordinates before $x()$ is applied..

d. $\varphi(x)$ maps field elements to non-negative integers, taking the bit vector representation of a field element and interpreting it as the binary expansion of an integer.

The precise definition of $\varphi(x)$ used in steps 6 and 7 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: $c_{m-1}2^{m-1} + \ldots + c_2 2^2 + c_1 2^1 + c_0\ \bmod p\ \in\ GF(p)\ ;$

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

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

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

Input:

1. *working_state*: The current values for *s, seedlen, p, a, b, n, P, Q, r_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.

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. *r_old*: The last output block.

5. *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. $t = s \oplus additional\_input.$

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

6. $s = \varphi(x(t * P)).$

Comment: *s* is a *seedlen*-bit number.

7. $r = \varphi(x(s * Q)).$

Comment: *r* is a *seedlen*-bit number.

Comment: Continuous test – Compare the old and new output blocks to assure that they are different.

8. If ($r = r\_old$), then return an **ERROR**.

9. $r\_old = r.$

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. $i = i + 1.$

67

14. If (**len** (*temp*) < *requested_number_of_bits*), then go to step 5.

15 *returned_bits* = **Truncate** (*temp, i × outlen, requested_number_of_bits*).

16. Return **SUCCESS**, *returned_bits,* and *s, r_old* 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.

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.
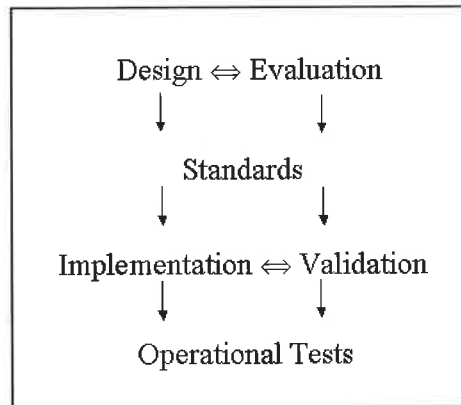
Figure 18: DRBG Assurance Strategy

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 [ebb12]: Page: 70
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., HMAC_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, 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 as specified in 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 Section 9.8).

11.4.2 Known-Answer Testing

Known-answer testing **shall** be conducted as specified in Section 9.7. A known-answer test involves operating the DRBG with data for which the correct output is already known and determining if the calculated output equals the expected output (the known answer). The test fails if the calculated output does not equal the known answer. In this case, the DRBG **shall** enter an error state and output an error indicator (see Section 9.8).

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:

$$y^2 = x^3 - 3x + b \ (\mathrm{mod}\ p)$$

Notation:

p - Order of the field F_p , given in decimal

r - order of the Elliptic Curve Group, in decimal . Note that r is used here for consistency with FIPS 186-3 but is referred to as n in the description of the **Dual_EC_DRBG (...)**

a – (-3) in the above equation

b - coefficient above

The x and y coordinates of the base point, ie generator G, are

the same as for the point P.

A.1.1.1 Curve P-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

72

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

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

A.1.1.2 Curve P-256

p = 11579208921035624876269744694940757353008614\

34152903141955336313088670978539 51

r = 11579208921035624876269744694940757352999695\

52241357603424222590610685120443 69

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

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

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

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

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

A.1.1.3 Curve P-384

p = 39402006196394479212279040100143613805079739\

27046544666794829340424572177149687032904726\

60882589380018616069731123 19

r = 39402006196394479212279040100143613805079739\

27046544666794690527962765939911326356939895\

63081522949135544336539426 43

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

Px = aa87ca22 be8b0537 8eb1c71e f320ad74 6e1d3b62 8ba79b98

73

```
        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 = 6864797660130609714981900799813932172694353\
    0014330540939446345918554318339765605212255 9\
    6406614545549772963113914808580371219879997 1\
    66438125740282911150571 51

r = 6864797660130609714981900799813932172694353\
    0014330540939446345918554318339765539424505 7\
    7463332171975329639963713633211138647686124 4\
    03803403728088927070054 49

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

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

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

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

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

74

Annex A.2 Using Alternative Points in the Dual_EC_DRBG()

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

A.2.1 Generating Alternative *P,Q*

The curve **shall** be one of the NIST curves from FIPS 186-3 that is specified in Annex A.1 of this Standard, and **shall** be appropriate for the desired *security_strength*, as specified in Table 5, Section 10.3.2.1.

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

The point Q **shall** be generated using the procedure specified as Algorithm A.2.4.3 from ANS X9.62 (Draft-2005-03-11). That algorithm requires the following input:

> An elliptic curve $E = (F_q, a, b)$, cofactor h, prime n, a bit string *SEED*, and hash function **Hash**(). The curve parameters are given in Annex A.1. The minimum length m of *SEED* **shall** conform to Section 10.3.2.1, Table 5, under "Seed length". The bit length of *SEED* may be larger than m. The hash function **shall** be SHA-512 in all cases.

> If the output of algorithm A.2.4.3 is "failure" a different *SEED* will have to be used.

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

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

To insure that the point Q has been generated appropriately, an additional self-test procedure **shall** be performed whenever the instantiate function is invoked. Section 9.7.2 specifies that known-answer tests on the instantiate function be performed prior to creating an operational instantiation. As part of those tests, an implementation of Algorithm A.2.4.3 **shall** be called with the *SEED* value used to generate the alternate Q. The point returned **shall** be compared with the stored value of Q used in place of the default value (see Annex A.1). If the generated value does not match the stored value, the implementation **shall** halt with an error condition.

ANNEX B : (Normative) Conversion and Auxilliary Routines

B.1 Bitstring to an Integer

Input:

1. $b_1, b_2, ..., b_n$ 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 $O_1, O_2, ..., O_n$ 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 $O_1, O_2, ..., O_n$ 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 to 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[3]. 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.

---

[3] 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 $2^{b \cdot 2}$ 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			$f \geq 512$

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.

80

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 "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 $2 * p^{\frac{1}{2}}$ 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 $f = 1$ 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 $d \ll m$.

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 $(m-d)$-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 $m\text{-}d$: 256

$\log2(f)$: 0 d: 1 entropy: 254.50000000 $m\text{-}d$: 255

$\log2(f)$: 0 d: 2 entropy: 253.78063906 $m\text{-}d$: 254

$\log2(f)$: 0 d: 3 entropy: 252.90244224 $m\text{-}d$: 253

$\log2(f)$: 0 d: 4 entropy: 251.95336161 $m\text{-}d$: 252

$\log2(f)$: 0 d: 5 entropy: 250.97708960 $m\text{-}d$: 251

$\log2(f)$: 0 d: 6 entropy: 249.98863897 $m\text{-}d$: 250

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

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

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

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

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

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

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

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

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

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

Observations:

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

    b) The missing entropy rapidly decreases;

    c) For $\log2(f) = 0$, i.e, the mod p curves, d=13 leaves 1 bit of information in every 10,000 (m-13)-bit outputs.

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

For ease of implementation, the value of d **should** be adjusted upward, if necessary, until the number of bits remaining , m-d= *outlen,* is a multiple of 8. By this rule, the recommended number of bits discarded from each x-coordinate will be either 16 or 17. As noted in Section 10.3.2.2.4, an implementation may decide to truncate additional bits from each x-coordinate, provided the number retained is a multiple of 8.

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

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

82

provide the following bound:

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

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

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

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

83

ANNEX D: (Informative) DRBG Selection

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

D.1 Choosing a DRBG Algorithm

Almost no application or system designer starts with the primary purpose of generating 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 set of cryptographic primitives as part of his implementation. By choosing a DRBG based on one of these primitives, he can minimize the cost of adding that DRBG. In hardware, this translates to lower gate count, less power consumption, and less hardware that must be protected against probing and power analysis. In software, this translates to fewer lines of code to write, test, and validate.

   For example, a module that generates RSA signatures has available some kind of hashing engine, so a hash-based DRBG is a natural choice.

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

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

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

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

D.2 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

84

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.

D.2.1 Hash_DRBG

▬ HMAC_DRBG

HMAC_DRBG is a DRBG whose security is based on the assumption that HMAC is a pseudorandom function. [I think the following needs to be either augmented to complete the ideas, or removed.] The security of **HMAC_DRBG** is based on an attacker getting sequences of no more than to 2^{35} bits, generated by the following steps:

$temp$ = the $Null$ string.

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.

D.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.

D.2.2.2 Performance Properties

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.

D.2.3 Summary and Comparison of Hash-Based DRBGs

D.2.3.1 Security

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

HMAC_DRBG:

$$V_1 = \text{HMAC}(K, V_0) = \text{Hash}(\text{opad}(K) \| (\text{Hash}(\text{ipad}(K) \| V_0)).$$

$$V_2 = \text{HMAC}(K, V_1) = \text{Hash}(\text{opad}(K) \| (\text{Hash}(\text{ipad}(K) \| V_1)).$$

$$V_3 = \text{HMAC}(K, V_2) = \text{Hash}(\text{opad}(K) \| (\text{Hash}(\text{ipad}(K) \| V_2)).$$

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[4]. (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.

---

[4] 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).

IHMAC_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.

D.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.

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

> Request overhead: six compression functions[5].

> 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[6].

> Cost for *outlen* bits of pseudorandom output: four compression functions[7].

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

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.

---

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

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

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

D.3 DRBGs Based on Block Ciphers

D.3.1 The Two Constructions: CTR and OFB

This standard describes DRBGs based on block ciphers using the CTR-mode. The CTR mode guarantees that short cycles cannot occur in a single output request. The security of the 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 the CTR_DRBG and a hash function-based DRBG, 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.

D.3.2 Choosing a Block Cipher

The choice of the block cipher algorithm to be used is a security issue. At present, only TDEA and AES are approved block cipher algorithms.

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 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.

    b. 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 $2^{security\ strength/2}$ 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

88

$2^{security\_strength/2}$ 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. 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 CTR_DRBG used with TDEA is 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.**

D.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.

89

D.4 DRBGs Based on Hard Problems

The **Dual_EC_DRBG** bases its 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).

This algorithm is 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.

D.4.1 Implementation Considerations

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

Because of the various security strengths 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 [ebb13]:** Page: 90
> 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. 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.

ANNEX E: (Informative) Example Pseudocode for Each DRBG

E.1   Preliminaries

The internal states in these examples are considered to be an array of states, identified by *state_handle*. A particular state is addressed as *internal_state* (*state_handle*), where the value of *state_handle* begins at 0 and ends at *n*-1, and *n* is the number of internal states provided by an implementation. A particular element in the internal state is addressed by *internal_state* (*state_handle*).*element*.

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

The following routine is defined for these pseudocode examples:

Find_state_space (): A function that finds an unused internal state. The function returns a *status* (either "Success" or a message indicating that an unused internal state is not available) and, if *status* = "Success", a *state_handle* that points to an available *internal_state* in the array of internal states. If *status* ≠ "Success", an invalid *state_handle* is returned.

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

E.2

E.3   HMAC_DRBG Example

E.3.1   Discussion

This example of HMAC_DRBG uses the SHA-256 hash function. Reseeding and prediction resistance are not provided. The nonce for instantiation consists of a random value with *security_strength*/2 bits of entropy; the nonce is obtained by increasing the call for entropy bits via the **Get_entropy** call by *security_strength*/2 bits (i.e., by adding *security_strength*/2 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 bits.

3. Required minimum entropy for the entropy input at instantiation = 3/2 *security_strength* (this includes the entropy required for the nonce).

4. Minimum entropy input length (*min _length*) = 3/2 *security_strength* (this includes the minimum length for the nonce).

5. Seed length (*seedlen*) = 440 bits.

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*) = 160 bits.

9. Maximum length of the entropy input (*max _length*) = 1000 bits.

E.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*)>160), then **Return** ("*Personalization_string* too long", -1)

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

3. If (*requested_security_strength* ≤ 112), then *security_strength* = 112

   Else (*requested_ security_strength* ≤ 128), then *security_strength* = 128

   Else (*requested_ security_strength* ≤ 192), then *security_strength* = 192

   Else *security_strength* = 256.

> Comment: Get the *entropy_input and the nonce*.

92

4. *min_entropy* = 1.5 × *security_strength*.

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

6. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).

7. *reseed_counter* = 1.

8. **Return** (*V*, *Key*, *reseed_counter*).

E.3.3 Generating Pseudorandom Bits Using HMAC_DRBG

The implementation returns a *Null* string as the pseudorandom bits if an error has been detected. This function uses the **Update** function specified in Section 10.1.3.2.2, and the **Uninstantiate** function in Section 9.5.

HMAC_DRBG(...):

Input: integer (*state_handle, requested_no_of_bits, requested_security_strength*).

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

Process:

> Comment: Check for a valid state handle.

1. If ((*state_handle* < 0) or (*state_handle* > 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* = "Reseed required"), then **Return** ("DRBG can no longer be used. Please re-instantiate or reseed", *Null*).

8. If (*status* = "ERROR: outputs match"), then

    8.1 For *i* = 0 to 3, do

        8.1.1 *status* = **Uninstantiate** (*i*).

        8.1.2 If (*status* ≠ "Success"), then **Return** ("DRBG FAILURE: Successive outputs match, and uninstantiate failed", *Null*).

    8.2 **Return** ("DRBG " || *status, Null*).

> Comment: Update the internal state.

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

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

Generate_algorithm (...):

Input: bitstring (*V_old, Key_old*), integer (*reseed_counter*,

94

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_old, V_old*).

   3.2 If (*V* = *V_old*), then **Return** ("ERROR: outputs match", *Null, V, Key, reseed_counter*).

   3.3 *V_old* = *V.*

   3.4 *temp* = *temp* ‖ *V.*

4. *pseudorandom_bits* = Leftmost (*requested_no_of_bits*) of *temp.*

5. (*Key, V*) = **Update** (*additional_input, Key_old, V_old*).

6. If ((*V* = *V_old*) or (*Key* = *Key_old*)), then **Return** ("ERROR: outputs match", *Null, V, Key, reseed_counter*).

7. *reseed_counter* = *reseed_counter* + 1.

8. **Return** ("Success", *pseudorandom_bits, V, Key, reseed_counter*).

E.4 CTR_DRBG Example Using a Derivation Function

E.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. 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, previous_output_block, reseed_counter*, and *security_strength*, where *V, Key* and *previous_output_block* are strings, and all other values are integers.Since prediction resistance is always available, there is no need for *prediction_resistance_flag* in the internal state.

In accordance with Table 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,

95

reseed and generate functions:

1. *highest_supported_security_strength* = 128.

2. Output block length (*outlen*) = 128 bits.

3. Key length (*keylen*) = 128 bits.

4. Required minimum entropy for the entropy input at instantiate and reseed = *security_strength*.

5. Minimum entropy input length (*min _length*) = *security_strength* bits.

6. Maximum entropy input length (*max _length*) = 1000 bits.

7. Maximum personalization string input length (*max_personalization_string_input_length*) = 800 bits.

8. Maximum additional input length (*max_additional_input_length*) = 800 bits.

9. Seed length (*seedlen*) = 256 bits.

10. Maximum number of bits per request (*max_number_of_bits_per_request*) = 4000 bits.

11. Reseed interval (*reseed_interval*) = 100,000 requests. Note that for this value, the instantiation count will not repeat during the reseed interval.

E.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*).

E.4.3   Instantiation of CTR_DRBG Using a Derivation Function

This implementation will return a text message and an invalid state handle (-1) when an error is encountered. **Block_Cipher_df** is the derivation function in Section 9.6.3, and uses AES-128 in ECB mode as the **Block_Encrypt** function.

Note that this implementation does not include the *prediction_resistance_flag* in the input parameters, nor save it in the internal state, since prediction resistance is always available.

Instantiate_CTR_DRBG (...):

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

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

> **Process:**

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

> 1. If (*requested_instantiation_security_strength* > 128) then **Return** ("Invalid *requested_instantiation_security_strength*", -1).

> 2. If (**len** (*personalization_string*) > 800), then **Return** ("*Personalization_string* too long", -1).

> 3. If (*requested_instantiation_security_strength* ≤ 112), then *security_strength* = 112

>   Else *security_strength* = 128.

>> Comment: Get the entropy input.

> 4. (*status*, *entropy_input*) = **Get_entropy** (*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 the nonce wraps when it's storage limit is reached, and that the counter pertains to all instantiations, not just this one.

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

>> Comment: Invoke the instantiate algorithm.

> 7. (*V*, *Key*, *previous_output_block*, *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*, *previous_output_block*, *reseed_counter*, *security_strength*}.

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. *first_output_block* = **AES_ECB_Encrypt** (*Key*, *V*).

8. *zeros* = $0^{seedlen}$.   Comment: Produce a string of *seedlen* zeros.

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

10. **Return** (*V*, *Key*, *first_output_block*, *reseed_counter*).

E.4.4 Reseeding a CTR_DRBG Instantiation Using a Derivation Function

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

Reseed_CTR_DRBG_Instantiation (...):

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

Output: string *status*.

Process:

Comment: Check for the validity of *state_handle*.

98

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

    Comment: Get the internal state values.

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

3. If (**len** (*additional_input*) > 800), then **Return** ("*Additional_input* too long").

4. (*status*, *entropy_input*) = **Get_entropy** (*security_strength*, *security_strength*, 1000).

6. If (*status* ≠ "Success"), then **Return** ("Failure indication returned by the entropy source:" ∥ *status*).

    Comment: Invoke the reseed algorithm.

7. (*status*, *V*, *Key*, *reseed_counter*) = **Reseed_algorithm** (*V*, *Key*, *reseed_counter*, *entropy_input*, *additional_input*).

8. If (*status* ≠ "Success"), then

    8.1 For *i* = 0 to 5, do

        8.1.1 *status* = **Uninstantiate** (*i*).

        8.1.2 If (*status* ≠ "Success"), then **Return** ("DRBG FAILURE: Successive outputs match, and uninstantiate failed", *Null*).

    8.2 **Return** ("DRBG:" ∥ *status*, *Null*).

    Comment: Save the new internal state.

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

10. **Return** ("Success").

Reseed_algorithm (...):

> **Input**: bitstring (*V_old*, *Key_old*), integer (*reseed_counter*), bitstring (*entropy_input*, *additional_input*).

> **Output**: string *status*, 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_old*, *_old*).

99

4. If ((Key = Key_old) or (V = V_old)), then **Return** (ERROR: updates match").

5. *reseed_counter* = 1.

6. Return ("Success", V, *Key*, *reseed_counter*).

E.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* < 0) or (*state_handle* > 5) or (*internal_state* (*state_handle*) = {*Null, 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*, *previous_output_block* = *internal_state* (*state_handle*). *previous_output_block*, *security_strength* = *internal_state* (*state_handle*).*security_strength*, *reseed_counter* = *internal_state* (*state_handle*).*reseed_counter*.

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

> 3. If (*requested_no_of_bits* > 4000), then **Return** ("Too many bits requested", *Null*).

> 4. If (*requested_security_strength* > *security_strength*), then **Return** ("Invalid *requested_security_strength*", *Null*).

> 5. If (**len** (*additional_input*) > 800), then **Return** ("*Additional_input* too long", *Null*).

> 6. *reseed_required_flag* = 0.

> 7. If (*reseed_required_flag* = 1), then

>> 7.1 *status* = **Reseed_CTR_DRBG_Instantiation** (*state_handle*, *additional_input*).

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

>>> Comment: Get the new working state values;

100

> the administrative information was not affected.

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

7.4   *additional_input = Null.*

7.5   *reseed_required_flag = 0.*

> Comment: Generate bits using the generate algorithm.

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

9.   If (*status* = "Reseed required"), then

     9.1   *reseed_required_flag = 1.*

     9.2   Go to step 7.

10. If (*status* = "ERROR: outputs match"), then

     10.1 For *i* = 0 to 5, do

         8.1.1   *status* = **Uninstantiate** (*i*).

         8.1.2   If (*status* ≠ "Success"), then **Return** ("DRBG FAILURE: Successive outputs match, and uninstantiate failed", *Null*).

     10.2 **Return** ("DRBG: " ‖ *status*, *Null*).

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

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

Generate_algorithm (...):

    Input: bitstring (*V_old, Key_old, previous_output_block*), integer (*reseed_counter, requested_number_of_bits*) bitstring *additional_input.*

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

    Process:

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

2.5   If((*Key* = *Key_old*) or (*V* = *V_old*)), then **Return** ("ERROR: outputs
       match", *Null, V, Key, previous_output_block, reseed_counter*).

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   If (*output_block* = *previous_output_block*), then **Return** ("ERROR:
          outputs match", *Null, V, Key, previous_output_block,
          reseed_counter*).

   4.4   *previous_output_block* = *output_block*.

   4.5   *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_old* = *Key*; *V_old* = *V*.

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

9. If ((*Key* = *Key_old*) or (*V* = *V_old*), then **Return** ("ERROR: outputs match",
   Null, V, Key, previous_output_block, reseed_counter).

10. *reseed_counter* = *reseed_counter* + 1.

11. **Return** ("Success", *returned_bits, V, Key, previous_output_block,
    reseed_counter*).

E.5   CTR_DRBG Example Without a Derivation Function

E.5.1   Discussion

This example of **CTR_DRBG** is the same as the previous example except that a derivation
function is not used (i.e., full entropy is always available). As before the CTR_DRBG uses
AES-128. The reseed and prediction resistance capabilities are available. Both a
personalization string and additional input are allowed. A total of 5 internal states are
available. For this implementation, the functions and algorithms are written as separate
routines. The **Block_Encrypt** function uses AES-128 in the ECB mode.

The nonce for instantiation (*instantiation_nonce*) consists of a 32-bit incrementing counter that is prepended 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*, *previous_output_block*, *reseed_counter*, and *security_strength*, where *V*, *Key* and *previous_output_block* are strings, and all other values are integers.Since prediction resistance is always available, there is no need for *prediction_resistance_flag* in the internal state.

In accordance with Table 4 in Section 10.2.1, security strengths of 112 and 128 may be supported. The definitions are the same as those provided in Annex E.4.1, except that the maximum size of the *personalization_string* is 224 bits in order to accommodate the 32-bits of the *instantiation_nonce* (i.e., **len** (*instantiation_nonce*) + **len** (*personalization_string*) must be ≤ *seedlen*). In addition, the maximum size of any *additional_input* is 256 bits (i.e., len (*additional_input* ≤ *seedlen*).

E. 5.2 The Update Function

The update function is the same as that provided in Annex E.4.2.

E.5.3 Instantiation of CTR_DRBG Without a Derivation Function

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

Note that this implementation does not include the *prediction_resistance_flag* in the input parameters, nor save it in the internal state, since prediction resistance is always available.

Instantiate_CTR_DRBG (...):

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

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

> **Process:**

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

> 1. If (*requested_instantiation_security_strength* > 128) then **Return** ("Invalid requested_instantiation_security_strength", -1).

> 2. If (**len** (*personalization_string*) > 224), 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 the nonce wraps when it's storage limit is reached, and that the counter pertains to all instantiations, not just this one.

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

Comment: Invoke the instantiate algorithm.

7. *personalization_string* = *instantiation_nonce* ‖ *personalization_string*.

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

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

Instantiate_algorithm (...):

Input: bitstring (*entropy_input, nonce, personalization_string*).

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

Process:

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

2. If (*temp* < 256), then *personalization_string* = *personalization_string* ‖ $0^{256-temp}$.

3. *seed_material* = *entropy_input* ⊕ *personalization_string*.

4. *Key* = 0^{128}. Comment: 128 bits.

5. *V* = 0^{128}.   Comment: 128 bits.

6. (*Key, V*) = **Update** (*seed_material, Key, V*).

7. $reseed\_counter = 1$.

8. $first_output_block = $ **AES_ECB_Encrypt** (Key, V).

9. $zeros = 0^{seedlen}$.   Comment: Produce a string of *seedlen* zeros.

10. $(Key, V) = $ **Update** $(zeros, Key, V)$.

11. **Return** $(V, Key, first\_output\_block, reseed\_counter)$.

E.5.4 Reseeding a CTR_DRBG Instantiation Without a Derivation Function

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

Reseed_CTR_DRBG_Instantiation (...):

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

    Output: string *status*.

    Process:

                Comment: Check for the validity of *state_handle*.

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

                Comment: Get the internal state values.

2. $V = internal\_state\ (state\_handle).V$, $Key = internal\_state\ (state\_handle).Key$, $previous\_output\_block = internal\_state\ (state\_handle).previous\_output\_block$, $security\_strength = internal\_state\ (state\_handle).security\_strength$.

3. If (**len** (*additional_input*) > 256), then **Return** ("*Additional_input* too long").

4. (*status, entropy_input*) = **Get_entropy** (*security_strength, security_strength*, 1000).

6. If (*status* ≠ "Success"), then **Return** ("Failure indication returned by the entropy source:" ‖ *status*).

                Comment: Invoke the reseed algorithm.

7. (*status, V, Key, reseed_counter*) = **Reseed_algorithm** (*V, Key, reseed_counter, entropy_input, additional_input*).

8. If (*status* ≠ "Success"), then

    8.1 For $i = 0$ to 5, do

        8.1.1 *status* = **Uninstantiate** (*i*).

105

> 8.1.2   If (*status* ≠ "Success"), then **Return** ("DRBG FAILURE:
> Successive outputs match, and uninstantiate failed", *Null*).

8.2   **Return** ("DRBG:" || *status*, *Null*).

> Comment: Save the new internal state.

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

10. **Return** ("Success").

Reseed_algorithm (...):

Input: bitstring (*V_old, Key_old*), integer (*reseed_counter*), bitstring (*entropy_input, additional_input*).

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

Process:

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

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

3. *seed_material = entropy_input* ⊕ *personalization_string*.

4. (*Key, V*) = **Update** (*seed_material, Key_old, _old*).

5. If ((*Key = Key_old*) or (*V = V_old*)), then **Return** (ERROR: updates match").

6. *reseed_counter* = 1.

7. Return ("Success", *V, Key, reseed_counter*).

E.5.5   Generating Pseudorandom Bits Using CTR_DRBG

The generate function is the same as that provided in Annex E.4.5.

E.6   Dual_EC_DRBG Example

E.6.1   Discussion

This example of **Dual_EC_DRBG** allows a consuming application to instantiate using any of the four prime 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 instantiation (*instantiation_nonce*) consists of a random value with *security_strength*/2 bits of entropy; the nonce is obtained by a separate call to the **Get_entropy** routine.

The internal state contains values for *s, seedlen, p, a, b, n, P, Q, r_old, block_counter* and *security_strength*. In accordance with Table 5 in Section 10.3.2.1, security strengths of

106

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 5.

3. Required minimum entropy for the entropy input at instantiation and reseed = *security_strength*.

4. Minimum entropy input length (*min _length*): See Table 5.

5. Maximum entropy input length (*max _length*) = 1000 bits.

6. Maximum personalization string length (*max_personalization_string_length*) = 800 bits.

7. Maximum additional input length (*max_additional_input_length*) = 800 bits.

8. Seed length (*seedlen*): See Table 5.

9. Maximum number of bits per request (*max_number_of_bits_per_request*) = 1000 bits.

10. Reseed interval (*reseed_interval*) = 10,000 blocks.

E.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. **Hash_df** is specified in Section 9.6.2.

Instantiate_Dual_EC_DRBG (...):

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

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

> **Process:**

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

> 1. If (*requested_instantiation_security_strength* > 256) then **Return** ("Invalid *requested_instantiation_security_strength*", -1).

> 2. If (**len** (*personalization_string*) > 800), then **Return** ("*personalization_string* too long", -1).

>> Comment : Select the prime field curve in accordance with the *requested_instantiation_security_strength*

> 3. If (*requested_instantiation_security_strength* ≤ 112), then

{*security_strength* = 112; *seedlen* = 224; *outlen* = 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. Select elliptic curve P-*seedlen* from Annex A to obtain the domain parameters p, a, b, n, P, and Q.

> Comment: Request *entropy_input*.

5. (*status*, *entropy_input*) = **Get_entropy** (*security_strength*, *min_entropy_input_length*, 1000).

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

7. (*status*, *instantiation_nonce*) = **Get_entropy** (*security_strength*/2, *security_strength*/2, 1000).

8. If (*status* ≠ "Success"), then **Return** ("Failure indication returned by the random nonce source:" ‖ *status*, -1).

> Comment: Perform the instantiate algorithm.

9. *seed_material* = *entropy_input* ‖ *instantiation_nonce* ‖ *personalization_string*.

10. s = **Hash_df** (*seed_material*, *seedlen*).

11. $r\_old = \varphi(x(s * Q))$.

12. *block_counter* = 0.

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

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

14. If (*status* ≠ "Success"), then **Return** (*status*, -1).

15. *internal_state* (*state_handle*) = {s, *seedlen*, p, a, b, n, P, Q, r_old, *block_counter*, *security_strength*}.

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

E.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* < 0) or (*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*) > 800), then **Return** ("Additional_input too long").

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

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

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

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

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

7. *internal_state* (*state_handle*).*s* = *s*.

8. *internal_state.block_counter* = 0.

9. **Return** ("Success").

E.6.4 Generating Pseudorandom Bits Using Dual_EC_DRBG

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

109

encountered.

Dual_EC_DRBG (...):

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

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

Process:

> Comment: Check for an invalid *state_handle*.

1. If ((*state_handle* < 0) or (*state_handle* > 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, P = internal_state* (*state_handle*).*P, Q = internal_state* (*state_handle*).*Q, r_old = internal_state* (*state_handle*).*r_old, block_counter = internal_state* (*state_handle*).*block_counter*.

> Comment: Check the rest of the input parameters.

3. If (*requested_number_of_bits* > 1000), then **Return** ("Too many bits requested", *Null*).

4. If (*requested_security_strength* > *security_strength*), then **Return** ("Invalid requested_strength", *Null*).

5. If (**len** (*additional_input*) > 800), then **Return** ("Additional_input too long", *Null*).

> Comment: Check whether a reseed is required.

6. If $\left(block\_counter + \left\lceil \dfrac{requested\_number\_of\_bits}{outlen} \right\rceil \right) > 10,000$, then

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

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

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

110

6.4   *additional_input* = *Null*.

> Comment: Execute the generate algorithm.

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

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

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

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

8.   *temp* = the *Null* string.

9.   *i* = 0.

10.   *t* = *s* $\oplus$ *additional_input*.

11. *s* = $\varphi(\, x(t \ast P))$.

12. *r* = $\varphi(\, x(s \ast Q))$.

13. If (*r* = *r_old*), then **Return** ("ERROR: outputs match", *Null)*.

14. *r_old* = *r*.

15. *temp* = *temp* || (**rightmost** *outlen* bits of *r*).

16. *additional_input*=$0^{seedlen}$.   Comment: *seedlen* zeroes; *additional_input* is added only on the first iteration.

17. *block_counter* = *block_counter* + 1.

18. *i* = *i* + 1.

19. If (**len** (*temp*) < *requested_no_of_bits*), then go to step 10.

20. *pseudorandom_bits* = **Truncate** (*temp*, *i* × *outlen*, *requested_no_of_bits*).

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

21. *internal_state.s* = *s*.

22. *internal_state.r_old* = *r_old*.

23. *internal_state.block_counter* = *block_counter*.

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

ANNEX F: (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].