8. Additional Requirements

8.1 General Discussion

In additional to the functional requirements in Section 7, other general requirements are levied on the implementation and use of a DRBG. These requirements are associated with the DRBG boundary, the state of a DRBG, the seed, any key that is used by a given DRBG, and any other input that is provided during operation. In addition, a discussion on prediction resistance and backtracking resistance in relation to the DRBGs specified in Section 10 is provided.

8.2 DRBG Boundary

DRBG processes **shall** be encapsulated within DRBG boundaries. A boundary may be either physical or conceptual. Within a DRBG boundary,

1. The DRBG internal state and the operation of the DRBG processes **shall** only be affected according to the DRBG specification.
2. The DRBG internal state **shall** exist solely within the DRBG boundary.
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. The internal state **shall** be contained within the DRBG boundary and **shall not** be accessible from outside the boundary.

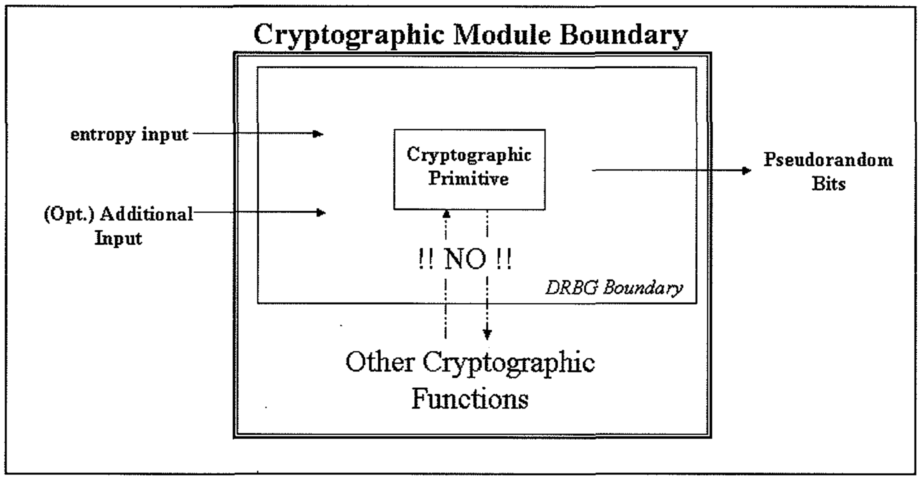

Figure 3: DRBG Boundary Contained within a Cryptographic Module Boundary

When a DRBG is implemented within a FIPS 140-2 cryptographic module, the DRBG boundary **shall** be either fully contained within the cryptographic boundary or **shall** be coincident with the cryptographic boundary of the DRBG.

Figure 3 depicts the DRBG boundary being fully contained within the cryptographic module boundary. The figure shows a generalized DRBG that contains a cryptographic primitive. This design will provide higher assurance of correct operation than a design with a coincident DRBG and cryptographic module boundary (see Figure 4). A cryptographic primitive within the DRBG boundary (e.g., a hash function) **shall not** be accessible for other purposes by a function outside the DRBG boundary; observe the !!NO!! across the dotted vertical arrows in the figure. For example, a digital signature function that is within the cryptographic boundary, but not within the DRBG boundary **shall not** use a function (e.g., the hash function) that resides within the DRBG boundary. In this case, a separate hash function is required outside the DRBG boundary, but within the cryptographic module boundary for digital signature purposes.

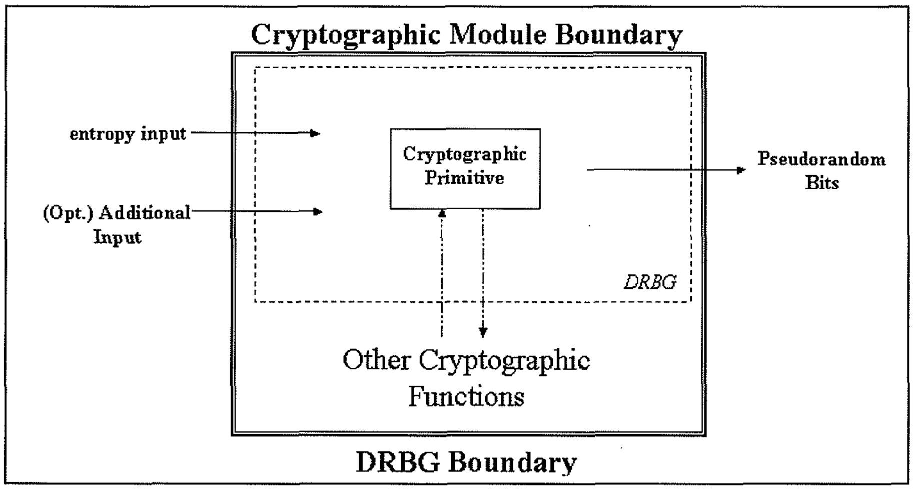

Figure 4: Coincident DRBG and Cryptographic Module Boundaries

Figure 4 depicts coincident DRBG and Cryptographic Module boundaries (i.e., the boundaries are identical). Note that the DRBG itself is contained within the dotted rectangle for convenience only; this is not intended to indicate the actual DRBG boundary. This design provides lesser assurance than the previous design because the internal state is potentially accessible by other non-DRBG functions. In this case, a cryptographic primitive within a DRBG boundary **may** be used by other cryptographic functions within the coincident DRBG and cryptographic module boundaries (e.g., during the generation or validation of digital signatures); observe the vertical dashed arrows to and from the DRBG's cryptographic primitive. However, the internal state of the DRBG **shall not** be used or affected by other non-DRBG functions within the coincident boundary.
In both figures, the entropy input is shown as being provided from outside the cryptographic module and DRBG boundaries. This is depicted as such for convenience only. The entropy input may actually be provided from either inside or outside the two

boundaries. In either case, however, the requirements for protecting and handling the entropy input and the resulting seed are specified in Section 8.5.

All DRBG processes need not be contained within the same DRBG or cryptographic module boundary. Particularly in the case of restricted environments (e.g., smart cards), it may be beneficial to distribute the DRBG processes. See Section 8.3 and Annex B for further discussion.

8.3 Model of DRBG Processes

A DRBG requires instantiation, generation, testing and uninstantiation processes. A DRBG **may** also include a reseeding process. Figure 5 depicts the use of a DRBG by an application that contains the full "suite" of DRBG processes.

Prior to requesting pseudorandom bits, the application **shall** instantiate the DRBG using a seed that is generated from entropy input. These seeds are used to determine the initial internal state of the DRBG instantiation. Although the entropy input is shown in the figure as originating outside the DRBG boundary, it may originate from within the boundary.

When the generation of pseudorandom bits is requested, the state is updated. Depending on the application, the DRBG instantiation may need to be periodically reseeded using new entropy input, either by a specific reseeding request or as determined by the generation process. During the reseeding process, new entropy input is obtained and a new internal state is determined.

When a DRBG instantiation is no longer required, the internal state may be "released" using an uninstantiate process.

An operational testing process **shall** be included within a DRBG boundary. The DRBG processes **shall** be tested at power-up, on-demand and at periodic intervals.

In actuality, a DRBG may be distributed. Figure 6 depicts the DRBG processes that **shall** be combined when DRBG processes are distributed. Other distribution

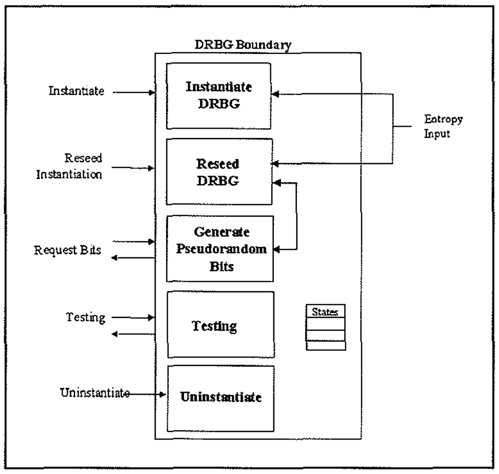

Figure 5: DRBG Processes

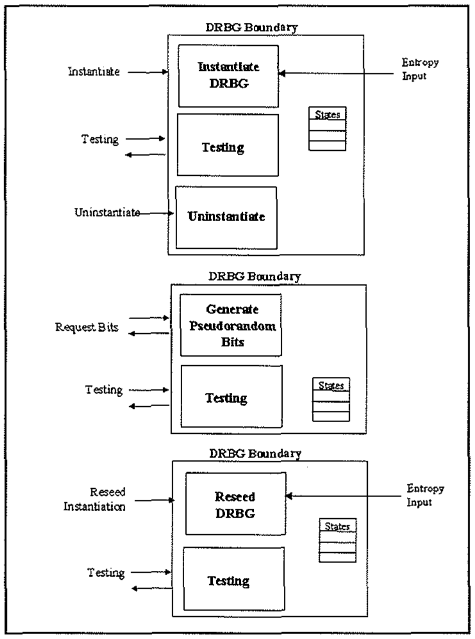

Figure 6: Distributed DRBG Processes

configurations are allowed, subject to the following:

1. Any DRBG boundary that includes an instantiation process **shall** include an uninstantiate and operational testing process.
2. A DRBG boundary containing a generation process **shall** include an operational testing process.
3. A DRBG boundary that contains a reseeding process **shall** include an operational testing process.

8.4 DRBG instantiation and the Internal State

A DRBG is instantiated for one or more purposes (see Section 9.4) by initializing with a seed; an instantiation may subsequently be reseeded. Each seed defines an instance of the DRBG instantiation; an instantiation consists of one or more instances that begin when a new seed is acquired (see Figure 7). The period of time between seeding and reseeding is considered as the seed life.

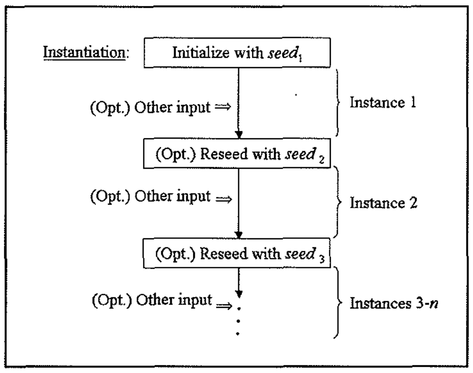

Figure 7: DRBG Instantiation

At any given time after a DRBG has been instantiated, a DRBG exists in a state that is defined by all prior input information. Different DRBG instances are defined by the seed and any other initial input information that is required by a specific DRBG.

A DRBG **shall** be instantiated prior to the generation of output by the DRBG.

During instantiation, an initial internal state (hereafter called just the state) is derived, in part, from a seed. The DRBG instantiation may subsequently be reseeded at any time (see Section 8.5 for a discussion on seeds).

Each DRBG instantiation will be associated with a different internal state. The state for an instantiation includes:

1. One or more values that are derived from the seed(s); at least one of these derived values is updated during the operation of the DRBG (e.g., at least one component of the state is updated during each call to the DRBG),
2. Other information that is particular to a specific DRBG; this information may remain static or may be updated during the operation of the DRBG,
3. An indication of whether or not prediction resistance is to be provided by the DRBG upon request,
4. The security strength provided by the DRBG, and
5. A transformation of the entropy input used to create the seed; this information remains static until replaced by new values during reseeding. This information need not be present if reseeding will not be performed.

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

A DRBG **shall** transition between states on demand (i.e., when the generator is requested to provide new pseudorandom bits). A DRBG **may** also be implemented to transition in response to internal or external events (e.g., system interrupts) or to transition continuously (e.g., whenever time is available to run the generator). Additional unpredictability is introduced when the generator transitions between states continuously or in response to external events. However, when the DRBG transitions from one state to another between requests, reseeding may need to be performed more frequently.

8.5 Seeds

8.5.1 General Discussion

When a DRBG is used to generate pseudorandom bits, a seed **shall** be acquired prior to the generation of output bits by the DRBG. The seed is used to instantiate the DRBG and determine the initial *state* that is used when calling the DRBG to obtain the first output bits.

The seed, seed size and the entropy (i.e., randomness) of the seed **shall** be selected to minimize the probability that the sequence of pseudorandom bits produced by one seed significantly matches the sequence produced by another seed, and reduces the probability that the seed can be guessed or exhaustively tested. Since this Standard does not require full entropy for a seed but does require sufficient entropy, the length of the seed may be greater than the entropy requirement (i.e., a seed with *n* bits of entropy may be longer than *n* bits in length).

The entry of entropy into a DRBG using an insecure method could result in voiding the intended security assurances. To ensure unpredictability, care **shall** be exercised in obtaining and handling the entropy input used to create seeds.

8.5.2 Generation and Handling of Seeds

The seed and its use by a DRBG **shall** be generated and handled as follows:

1. Seed construction: A seed **shall** include entropy input and **should** include a personalization string (see Figure 8). Note that it is possible, in some cases, that the entropy in the entropy input may not be distributed across the sequence of entropy input bits. Whether or not the personalization string is present, the resulting seed **shall** be unique. That is, when a personalization string is used, the combination of the entropy input and the personalization string **shall** determine a unique seed; when a personalization string is not used, the entropy input **shall** be statistically unique.
   The combination of the entropy input and the optional personalization string

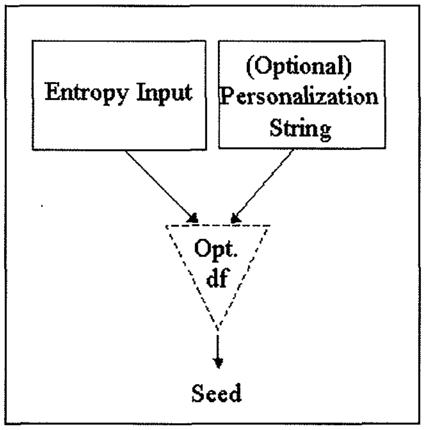

Figure 8: Seed Construction

is called the *seed material*. A derivation function **shall** be used to distribute the entropy in the entropy input across the entire seed (e.g., so that the seed is not constructed with all the entropy on one end of the seed) whenever:

- A personalization string is used, or
- A personalization string is not used, and the entropy in the entropy input is not independent and uniformly distributed throughout the entropy input string.

2. Entropy requirements: The entropy input for the seed **shall** contain sufficient entropy for the desired level of security, and the entropy **shall** be distributed across the seed (e.g., by an appropriate derivation function). The DRBGs **shall** have the required entropy provided in the entropy input. Additional entropy **may** be provided in a personalization string, but this is not required.
   A consuming application may or may not be concerned about collision resistance between seeds. In order to accommodate possible collision concerns, the entropy input for a seed **shall** have entropy that is equal to or greater than 128 bits or the required security strength for the consuming application, whichever is greater (i.e., $entropy \geq \mathbf{max}\ (128, security\_strength)$).
   Table 1 identifies the five security strengths to be provided by Approved DRBGs, along with the associated entropy requirements. If a selected DRBG and the entropy input for the seed are not able to provide the required strength required by the consuming application, then a different DRBG and entropy input **shall** be used.

Table 1: Minimum Entropy and Seed Size

Bits of Security Strength	80	112	128	192	256
Minimum entropy in the entropy input	128	128	128	192	256

3. Seed size: The minimum size of the seed depends on the DRBG and the security strength required by the consuming application. See Section 10 and Annex C.
4. 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 another source whose entropy characteristics are known. Further discussion about the entropy input is provided in Section 7.2.3. When sufficient entropy is not readily obtainable for multiple requests for entropy input (e.g., multiple seeds are required), but sufficient entropy is

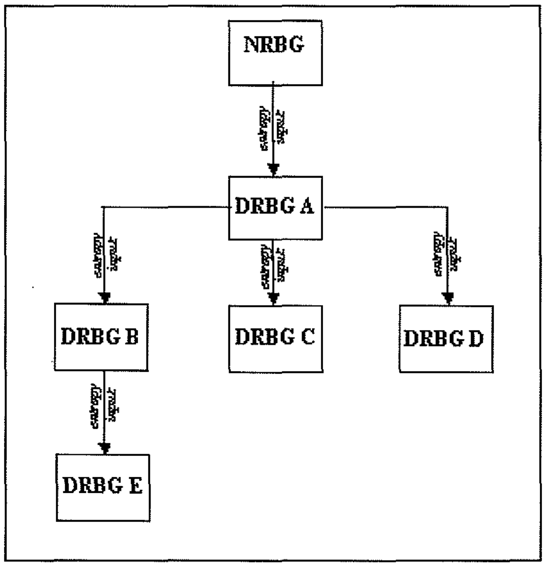

Figure 9: DRBG Chain

available for a single DRBG (e.g., DRBG A), this DRBG **may** be used to provide entropy input for other DRBGs (see Figure 9). In this case, the entropy provided to the first DRBG (i.e., DRBG A) **shall** be equal to or greater than the entropy requirement of any lower level DRBG. For example, DRBG A could provide entropy input for DRBGs B, C and D. The highest level DRBG (i.e., DRBG A) **may**, in fact, be used to provide entropy input for a chain of DRBGs. For example, DRBG A could provide entropy input for DRBG B, which in turn could be used to provide entropy input for DRBG E.

An entropy input source need not be co-located with the DRBG instantiation process. The entropy input could be provided to the instantiation process and combined with any personalization string to produce the seed and instantiate the DRBG. See Annex B for further discussion.

5. 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 only secrets in a cryptographic system are the keys, then the entropy inputs and seeds used to generate keys **shall** be treated as if they are keys.
6. Reseeding: Reseeding (i.e., replacement of one seed with a new seed) is a means of recovering the secrecy of the output of the DRBG if a seed 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).
   Generating too many outputs from a seed (and other input information) may provide sufficient information for successfully predicting future outputs unless prediction resistance is provided (see Section 8.8). 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 **shall** have a specified finite seedlife. A seed **shall** be replaced periodically. This **shall** be accomplished by 1) an explicit reseeding of the DRBG (e.g., by the application), or 2) by specifying prediction resistance when instantiating the DRBG and requesting the generation of pseudorandom bits (see Sections 9.5 and 9.7) or 3) by making the DRBG inoperable at the end of the seedlife. If entropy input and the seed become known (i.e., the seed is compromised), unauthorized entities may be able to determine the DRBG output.
   Reseeding of the DRBG (i.e., creating a new DRBG instance) **shall** be performed in accordance with the specification for the given DRBG. The DRBG reseed specifications within this standard are designed to produce a new seed that is determined by both the old seed and newly-obtained entropy input that will support the desired security level. The newly-obtained entropy input **shall** be checked to assure that it is not the same as the entropy input obtained to create the previous DRBG instance. More than one entropy input **shall not** be saved by the DRBG. The entropy input **shall not** be saved in its original form, but **shall** be transformed

by a one-way process (see the specifications in Section 10). When new entropy input is generated and compared to the "old" entropy input (i.e., the new entropy input is transformed and compared with the transformed old entropy input), the transformed new entropy input **shall** replace the old transformed entropy input in memory. If the new entropy input is determined to be identical to the old entropy input, then the DRBG **shall** fail.
It should be noted that an alternative to reseeding is to create an entirely new instantiation. This may be appropriate, for example, in environments with restricted capabilities, where the seed is obtained from a source that is not co-located with the DRBG (e.g., in a smart card applicaton).

7. 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 state has been initialized.
8. Seed separation: Seeds used by DRBGs **shall not** be used for other purposes (e.g., domain parameter or prime number generation).
It is recommended that when resources permit (e.g., storage capacity), different (i.e., statistically unique) seeds **should** be used for the generation of different types of random data (i.e., the instantiations of the DRBGs **should** be different). For example, the seed used to generate public values **should** be different than the seed used to generate secret values. The seed used by a DRBG technique to generate asymmetric key pairs **should** be different than a seed used by the same (or a different) DRBG technique to seed other DRBGs, which **should**, in turn, be different than a seed used by the same (or a different) DRBG technique to generate symmetric keys. The seed used by a DRBG technique to generate random challenges **should** be different than the seed used by the same (or a different) DRBG technique to generate PINS or passwords. However, the amount of seed separation is a cost/benefit decision.

Comment [ebb1]: Page: 50
John raised a basic question of whether we can use the same entropy input for multiple instantiations, but with different personalization strings. In this case, the seeds would be different.

8.6 Keys

Some DRBGs require the use of one or more keys. Such DRBGs are designed to derive keys from seeds (see Section 8.5, item 1 for a discussion on seed construction). A key and its use in a DRBG **shall** conform to the following:

1. Key entropy: The seed for the key **shall** have entropy that is equal to or greater than 128 bits or the required security strength of the consuming application, whichever is greater (i.e., $entropy \geq \max(128, security\_strength)$).
2. Key size: Key sizes **shall** be selected to support the desired security strength of the consuming application (see SP 800-57). If the DRBG primitive using the key (e.g.,

the block cipher algorithm) cannot support the required security strength, then a different primitive or a different DRBG **shall** be used.

3. Entropy input source for a key: The entropy input source for the key is the seed for the DRBG instance (see Section 8.5, item 4).
4. Key secrecy: Keys **shall** remain secret and **shall** be handled in a manner that is consistent with the security required for the data protected by the consuming application using the DRBG pseudorandom bits. Keys **shall** be protected in accordance with [SP 800-57].
5. Rekeying: Rekeying (i.e., replacement of one key with a new key) is a means of recovering the secrecy of the output of the DRBG if a key becomes known. Periodic rekeying is a good countermeasure to the potential threat that the keys and DRBG output become compromised. In some implementations (e.g., smartcards), an adequate rekeying process may not be possible. In these cases, the best policy might be to replace the DRBG, obtaining a new key in the process (e.g., obtain a new smart card).
   Generating too many outputs using a given key may provide sufficient information for successfully predicting future outputs when prediction resistance is not provided. Periodic rekeying will reduce security risks, reducing the likelihood of a compromise of the data that is protected by consuming applications that use the DRBG.

   Keys **shall** have a specified finite keylife (i.e. a cryptoperiod). Keys **shall** be replaced when seeds are replaced. This **shall** be accomplished as specified in Section 8.5, item 6. Expired keys or keys that have been replaced **shall** be destroyed (see SP 800-57). If keys become known (e.g., the keys or seeds are compromised), unauthorized entities may be able to determine the DRBG output.

6. Key use: Keys **shall** be used as specified in a specific DRBG. A DRBG requiring a key(s) **shall not** provide output until the key(s) is available.
7. Key separation: A key used by a DRBG **shall not** be used for any purpose other than pseudorandom bit generation. Different instantiations and different instances of the same instantiation of a DRBG **shall** use different keys.

8.7 Other Input

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 to derive a seed (see Section 8.5, item 1). When pseudorandom bits are requested and when reseeding is performed, additional input may be provided.

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.

8.7.1 Personalization String

A seed **should** be derived from both entropy input with sufficient entropy and a personalization string (see Section 8.5). That is, the use of a personalization string is good practice, but is not mandatory. The intent of a personalization string is to have information in the seed that differentiates one DRBG's seed from another DRBG's seed in order to increase assurance that two DRBG seeds are not inadvertently the same. Examples of data that may be included in a personalization string include a product and device number, user identification, date and timestamp, IP address, or any other information that helps to differentiate DRBGs, including secret information containing entropy.

8.7.2 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 length and value are arbitrary (i.e., there are no restrictions on its length or content). The additional input allows less reliance on both the seed and an entropy input source. If the additional input is kept secret and has sufficient entropy, the input may be used to provide additional entropy for random bit generation and provide an ability to recover from the compromise of the seed or one or more states of the DRBG.

8.8 Prediction Resistance and Backtracking Resistance

Figure 10 depicts the sequence of DRBG states that result from a given seed. Some subset of bits from each state are used to generate pseudorandom bits upon request by a user. The following discussions will use the figure to explain backtracking and prediction resistance. Suppose a compromise occurs at *State*$_X$, there *State*$_x$ contains both secret and public information.

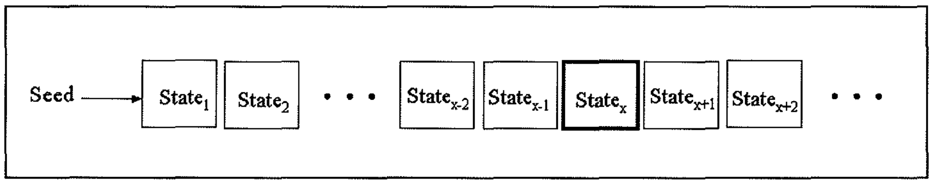

Figure 10: Sequence of DRBG States

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

For example, suppose that an adversary knows *State*$_{X}$ ~~and also knows the output bits from *State*$_1$ to *State*$_{x-2}$.~~ Backtracking resistance means that:

a. The output bits from *State*$_1$ to *State*$_{x-1}$ cannot be distinguished from random.

Formatted: Bullets and Numbering

Formatted

~~a.~~ b. The prior state values themselves (*State*$_1$ to *State*$_{x-1}$) cannot be 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*$_{x-2}$.~~

Formatted

Comment [ebb2]: Page: 52
This makes the definition very convoluted.

Backtracking resistance can be provided by ensuring that the state transition function of a DRBG is a one-way function[1], or by regenerating an additional[2] DRBG state from pseudorandom outputs at the end of each DRBG request.

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

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

a. The output bits from *State*$_{x+1}$ and forward cannot be distinguished from an ideal random bitstring by the adversary.

~~b.~~ b. The future 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*$_{x+n}$.~~

Formatted

Formatted: Bullets and Numbering

Formatted

Comment [ebb3]: Page: 53
This makes the definition very convoluted.

Prediction resistance can be provided only by ensuring that a DRBG is effectively reseeded between DRBG requests. That is, an amount of entropy sufficient to support the security level of the DRBG (i.e., for *strength* bits of security, *entropy* = **max** (128, strength)) must be added to the DRBG in a way that ensures that knowledge of the current~~previous~~ DRBG state does not allow an adversary any useful knowledge about future DRBG states or outputs. Note that inserting less than the required amount of entropy may improve the security of the DRBG, but does not guarantee prediction resistance.

[1] A one-way function is a function whose result is easy to compute, but extremely difficult to reverse. For example, for the function is $f(x) = y$, the result (y) is easy to compute given x and the function f. However, if f is a one-way function, and y is known, it is computationally infeasible to determine the value of x.

[2] Each DRBG is always updated after each request for pseudorandom bits. Prediction resistance would be provided by an additional update of the DRBG state.

9 General Discussion of the Specified DRBGs

9.1 Introduction

Numerous concepts have been employed in the DRBG specifications in Section 10. The following subsections are intended to provide an understanding of these concepts and how they are used.

9.2 Security Strength Supported by a DRBG Instantiation

The DRBGs specified in this Standard support one or more of five security strengths (i.e., security levels): 80, 112, 128, 192 or 256 bits. The security strengths that may be supported by a particular DRBG are specified for each. However, the security strength actually supported by a particular instantiation may be less than the maximum security strength possible for that DRBG, depending upon the amount of entropy that is contained in the seed.

The maximum strength provided by an instantiation is determined when the DRBG is instantiated. The instantiated security strength **shall** be less than or equal to the maximum security strength that can be supported by the DRBG.

For each DRBG instantiation, a security strength (i.e., security level) needs to be requested and obtained during the instantiation process. The DRBGs in Section 10 allow security strengths up to 256 bits, providing that the appropriate cryptographic primitives and sufficient entropy are available. Any security strength up to 256 **may** be requested. However, a DRBG will only be instantiated for one of the following five security levels: 80, 112, 128, 192 or 256. A requested security level that is between two of the five levels will be instantiated to the next highest level (e.g., a request for 96 bits of security will actually be instantiated at 112 bits of security).

When a DRBG instantiation needs to provide pseudorandom bits for only one purpose, then the security level needs to support that purpose. Examples:

1. 256-bit AES keys can provide a maximum of 256-bits of security. An instantiation must support at least 256 bits of security if the full 256 bits of security are to be provided by the AES keys. Note that the minimum entropy requirement would be 256 bits to support 256 bits of security.

2. 1024-bit DSA private keys can only provide 80 bits of security. In this case, an instantiation used only for the generation of 1024-bit DSA keys must be supported by at least 128 bits of entropy (see Section 9.3) and a DRBG that provides at least 80 bits of security.

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 entropy requirement (see Section 9.3). For example, if one purpose requires 80 bits of security (i.e., *min_entropy* = 128 bits), and another purpose requires 256 bits of security (i.e., *min_entropy* = 256 bits), then the DRBG **shall** be instantiated to support at least 256 bits of security (i.e., *min_entropy* = 256 bits).

9.3 Security Strength, Entropy and Seed Size of an Instantiation

The instantiation of a DRBG requires the generation of a seed with sufficient entropy to support the requested security strength; reseeding the instantiation requires the generation of another seed with the same properties. As discussed in Section 8.5, reseeding requires the acquisition of the appropriate amount of new entropy to support the desired security level and combining the newly-obtained entropy with the entropy from the previous instance.

As stated in Section 8.5, the minimum entropy (*min_entropy*) to be acquired when seeding or reseeding **shall** be equal to either 128 or the instantiated strength, whichever is greater (i.e., *min_entropy* = **max** (128, *strength*). Note that the use of more entropy than the minimum value will offer a security "cushion".

The minimum size of the *seed* depends on the DRBG. Many DRBGs allow a range of seed sizes. A variation in the allowable seed size permits the use of an entropy input source that provides either full entropy (i.e., one bit of entropy for each bit of the *seed*) or less than full entropy (i.e., multiple bits of the *seed* may be required to provide each bit of entropy).

9.4 DRBG Purposes and States

A DRBG may be used to obtain pseudorandom bits for different purposes (e.g., DSA private keys and AES keys). This Standard recommends that different instantiations be used to generate bits for different purposes. However, if an application needs to generate bits for different purposes, it may not always be practical to use multiple instantiations. Each instantiation is associated with an internal *state* for the purpose(s)supported by the instantiation. For example, a *state* may be associated with the generation of only 1024-bit DSA keys, and a separate *state* may be associated with the generation of 128-bit AES keys. Both *states* may use the same type of DRBG, but use different instantiations, or they may use different DRBG types (e.g., the generation of DSA keys may use the **Hash_DRBG (...)**, while AES keys may be generated using the **Dual_EC_DRBG (...)**). As another example, if an application cannot support multiple instantiations (e.g., because of memory restrictions), then the same internal *state* could be associated with generating both 1024-bit DSA keys and 128-bit AES keys (i.e., the *state* supports two purposes). As a third example, the same DRBG instantiation might be used for similar purposes (e.g., the generation of all digital signature keys, irrespective of the digital signature algorithm used).

A DRBG implementation may be designed to handle multiple instantiations. Sufficient "state space" **shall** be available for each instantiation, i.e., sufficient memory **shall** be available to store the internal state associated with each instantiation. In addition, within each DRBG boundary, state space **shall** always be available for operational testing. That is, sufficient memory **shall** always be available to perform operational testing without affecting the internal states associated with normal operation. For example, when a DRBG boundary contains the pseudorandom bit generation process, and the DRBG is intended to allow three separate instantiations, then state space for four internal states **shall** be allocated; at least one of these states **shall** always be available for operational testing.

9.5 Instantiating a DRBG

9.5.1 The Instantiation Function Call

Prior to the first request for pseudorandom bits, a DRBG **shall** be instantiated using a form of the following function call:

(*status*, *state_pointer*) = **Instantiate_DRBG** (*requested_strength*, *prediction_resistance_flag*, *personalization_string*, *DRBG_specific_parameters*, *mode*)

where:

1. *status* is the indication returned from the instantiation process. A *status* of Success indicates that the instantiation has been successful, and pseudorandom bits may be requested. Failure messages that could be returned from this process are specified for each DRBG. The *status* **shall** be checked to determine that the DRBG has been correctly instantiated.

2. *state_pointer* is used to identify the internal state for this instantiation in subsequent calls to the generation and reseed processes.

3. **Instantiate_ DRBG (...)** is specified for each DRBG. Note that the name of the generalized function call of this section (i.e., **Instantiate_DRBG (...)**) is different than the specific name used for each DRBG (e.g., **Instantiate_Dual_EC_DRBG (...)**).

4. *requested_strength* is used to request the minimum security strength for the instantiation. Note that DRBG implementations that support only one security strength do not require this parameter; however, any application using the DRBG must be aware of this limitation.

5. The *prediction_resistance_flag* indicates whether or not prediction resistance may be required by the consuming application during one or more requests for pseudorandom bits. Note that 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.

6. The *personalization_stri*ng is an optional input that is used to personalize a seed (see Section 8.4, item 1 and Section 8.7). If an implementation never intends to use a personalization string, then the parameter may be omitted.

7. The *DRBG_specific_parameters*, if any, are provided in Section 10 for each DRBG.

8. *mode* is used to indicate whether the instantiation is for normal operation or for testing.

9.5.2 Request for Entropy

The DRBG specifications in this Standard request bits from an entropy input source during the instantiation and reseeding processes and in order to provide prediction resistance. This is specified in each specification as:
(*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_length*, *max_length*, *mode*),
where

1. *status* is the status returned from the entropy input source. In the DRBG specifications, either an indication of Success or Failure is expected as the returned *status*. The *status* **shall** be checked to determine that the requested entropy input has been provided.
2. *entropy_input* is the string of bits returned from the entropy input source when the *mode* indicates normal operation, and the returned *status* = Success. For example, entropy_inputx might be used as the *seed* or used to derive the *seed*, depending on the DRBG. If the returned *status* = Failure, a Null string **shall** be returned. If the *mode* indicates a test is being performed, and *status* = Success, then a fixed value is returned as the *entopy_input* (see Section 9.9).
3. *min_entropy* is the minimum amount of entropy to be returned in the *entropy_input*. If an implementation always requires the same minimum entropy, this parameter may be omitted.
4. *min_length* is the minimum length of the bit string to be returned as the *entropy_input*. Note that *min_length* is determined either by the value of *min_entropy* or by the DRBG design requirements. If an implementation always requires the same minimum length, this parameter may be omitted.
5. *max_length* is the maximum length of the bit string to be returned as the *entropy_input*. Some of the DRBGs have a maximum length requirement in their design. Other DRBGs have no such restriction. If an implementation always requires the same maximum length, this parameter may be omitted.
6. *mode* indicates whether the request is made as a part of normal operation, in which actual entropy input bits are requested, or whether a test is being performed, in which case the *mode* is used to indicate what is being tested (see Section 9.9 for further details).

For implementations where the *min_length* is always the same as the *max_length*, the two parameters may be expressed as a single parameter (e.g., the call would be (*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *length*)).

The specific details of the **Get_entropy (...)** process are left to the implementer, with the above restrictions and any other entropy input source requirements in this Standard (see Sections 7.2.1, 8.5 and 8.6).

9.5.3 Find State Space

When a DRBG is instantiated, an area to save the internal state for that instantiation is required. The **Find_state_space (...)** function is called as follows:

(*status*, *state_pointer*) = **Find_state_space** (*mode*)

where

1. *status* is the status return from the **Find_state_space (...)** function. Either an indication of Success or Failure is expected. The *status* **shall** be checked to determine that the requested space has been allocated.
2. *state_pointer* is used to identify the internal state for this instantiation in subsequent calls to the generation and reseed processes.
3. *mode* indicates whether the request is made as part of normal operation or for operational testing.

The following or an equivalent process **shall** be used as the **Find_state_space (...)** function. Let *state_space* be an array of *n* internal states, and let the *state_space* array be numbered from 0 to *n*-1. Let *Empty* represent an empty state (i.e., a state space that has not been assigned to an instantiation). The actual value of *Empty* is DRBG dependent.

Find_state_space (...):

Input: integer *mode*.

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

Process:

```
                                   Comment: only allow the last
                                   state_space to be used for testing;
                                   otherwise, testing may use any
                                   available state_space.
1. If (mode = Normal_operation), then last_state = n-2
   Else last_state = n-1.
                                   Comment: Search for an empty
                                   state_space.
2. For i = 0 to last_state do
   If (state_space (i) = Empty), then Return ("Success", i).
3. Return ("No available state space", Invalid_state_pointer).
```

9.5.4 Derivation Functions

9.5.4.1 Introduction

Derivation functions are used during DRBG instantiation and reseeding to either derive state values or to distribute entropy throughout a bit string. Two methods are provided. One method is based on hash functions and the other method is based on the block cipher algorithm used by a given DRBG.

9.5.4.2 Derivation Function Using a Hash Function

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. Note that the *requested_bits* string **shall not** be greater than (255 × *outlen*) bits in length (i.e., *no_of_bits_to_return* ≤ (255 × *outlen*)). However, implementations may use a smaller value (*max_no_of_bits*) whose value is an implementation choice that **shall** be ≤ (255 × *outlen*)). The following or an equivalent process **shall** be used to derive the requested number of bits.

Hash_df (...):

```
Input: bitstring input_string, integer no_of_bits_to_return.
Output: bitstring requested_bits.
Process:
  1. If (no_of_bits_to_return > max_no_of_bits), then Return ("Too many bits
     requested from derivation function").
  2. temp = the Null string.
  3. $len = \left\lceil \frac{no\_of\_bits\_to\_return}{outlen} \right\rceil$.
  4. counter = an 8 bit binary value represented in hexadecimal as x'01'.
  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 (requested_bits).
```

Comment [ebb4]: Page: 60
We need to specify an integer to string conversion process.

9.5.4.3 Derivation Function Using a Block Cipher Algorithm

9.5.4.3.1 The TDEA_df (...)Derivation Function

The **TDEA_df (...)** function derives bits from an input string using the TDEA block cipher algorithm, a derivation key and an Approved key wrapping algorithm (**TDEA_Wrap (...)**). **TDEA_Wrap(...)** is define in ANSI X9.102. Note that two key and three key TDEA are specified. Two keys are presented in a 112-bit string; three keys are presented in a 168-bit string.

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

TDEA_df (...):

```
Input: integer keylen, bitstring (derivation_key, M), integer no_of_bits_to_return.
Output: string status, bitstring (requested_bits).
```

Process:

```
                                          Comment: Parse the *derivation_key*
                                          into three TDEA keys (see below).
    1. (status, key1, key2, key3) = Parse_TDEA_Key (derivation_key).
    2. If (status = "Failure"), then Return ("Invalid Key size", Null).
                                          Comment: Wrap M using the three
                                          TDEA keys; the ciphertext string is
                                          returned as C.
    3. If (no_of_bits_to_return > len (M)), then Return ("Too many bits requested
       from TDEA_df", Null).
    4. C = TDEA_Wrap (key1, key2, key3, (no_of_bits_to_return || M)).
    5. requested_bits = Leftmost (no_of_bits_to_return) of C.
    6. Return ("Success", requested_bits).
```

Comment [ebb5]: Page: 60
We need to specify an integer to string conversion process.

Parse_TDEA_Key (...):

```
  Input: bitstring Key.
  Output: string status, bitstring (key_1, key_2, key_3).
    1. keylen = len (Key).
    2. If (keylen = 112), then do:
       2.1  key_1 = Leftmost 56 bits of Key.
       2.2  key_2 = Rightmost 56 bits of Key.
       2.3  key_3 = key_1.
       2.4  Return ("Success", key_1, key_2, key_3).
    3. If (keylen = 168), then do:
       3.1  key_1 = Leftmost 56 bits of Key.
       2.2  key_2 = Bits 57-112 of Key.
       2.3  key_3 = Rightmost 56 bits of Key.
       2.4  Return ("Success", key_1, key_2, key_3).
    4. Return ("Failure").
```

9.5.4.3.2 The AES_df (...)Derivation Function

The **AES_df (...)** function derives bits from an input string using the AES block cipher algorithm, a derivation key and an Approved key wrapping algorithm (**AES_Wrap (...)**). **AES_Wrap (...)** is define in ANSI X9.102. Note that AES keys may consist of 128, 192 or 256 bits.

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

AES_df (...):

```
  Input: integer keylen, bitstring (derivation_key, M), integer no_of_bits_to_return.
  Output: string status, bitstring requested_bits.
  Process:
    1. If (no_of_bits_to_return > len (M)), then Return ("Too many bits requested
       from AES_df", Null).
                                          Comment: Get the ciphertext string C.
    2. C = AES_Wrap (derivation_key, keylen, (no_of_bits_to_return || M)).
    3. requested_bits = Leftmost (no_of_bits_to_return) of C.
```

Comment [ebb6]: Page: 62
Need to specify an integer to string conversion process.

4. **Return** ("Success", *requested_bits*).

9.6 Reseeding a DRBG Instantiation

9.6.1 Introduction

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 process. Reseeding may be :

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

Alternatively, a new DRBG instantiation may be created (see Section 9.5).

During reseeding, a DRBG **shall not** continue to produce output bits until the DRBG is completely reseeded and a new internal state is determined.

9.6.2 The Function Call

When a DRBG instantiation is reseeded, the DRBG **shall** be reseeded using a form of the following function call:

status = **Reseed_ DRBG_Instantiation** (*state_pointer*, *additional_input*, *mode*)

where:

1. *status* is the indication returned from the reseeding process. A *status* of Success indicates that the reseeding process has been successful, and pseudorandom bits may be requested. Failure messages that could be returned from this process are specified for each DRBG. The *status* **shall** be checked to determine that the DRBG has been correctly reseeded.
2. **Reseed__DRBG_Instantiation (...)** is specified for each DRBG. Note that the name of the generalized function call of this section (i.e., **Reseed_DRBG_Instantiation (...)**) is different than the specific name used for each DRBG (e.g., **Reseed_Hash_DRBG_Instantiation (...)**).
3. *state_pointer* indicates the internal state to be reseeded.
4. Optional *additional_input* may be provided. This parameter is not required for implementations that will never use the additional input.
5. *mode* is used to indicate whether the reseeding is for normal operation or for testing.

9.7 Generating Pseudorandom Bits Using a DRBG

9.7.1 Introduction

Each request for pseudorandom bits **shall** generate bits for only one value. For example, a single request **shall not** be used to generate bits for multiple AES keys, or bits for both an AES key and a DSA key). Instead, separate calls to the generation function **shall** be used.

Multiple requests **may** be used to construct a single value. For example, a 1024 bit pseudorandom string may be generated using eight calls for 128 bits each, and concatenating the eight 128-bit strings.

9.7.2 The Function Call

An application may request the generation of pseudorandom bits by a DRBG using a form of the following call:

> (*status*, *pseudorandom_bits*) = **DRBG** (*state_pointer*, *requested_no_of_bits*, *requested_strength*, *additional_input*, *prediction_resistance_flag*, *mode*)

where:

1. *status* is the indication returned from **DRBG (...)**. A *status* of Success indicates that *pseudorandom_bits* have been successfully generated. Failure messages that could be returned from this process are specified for each DRBG. If an indication of failure is returned, a Null string is returned in place of the *pseudorandom_bits*. The status returned by the DRBG **shall** be checked by the consuming application to determine that the request has been successful prior to using any bit string returned.

2. *pseudorandom_bits* are returned when the *status* indicates Success. These are the bits requested by the application. If the status indicates a failure, a Null string **shall** be returned.

3. **DRBG (...)** is specified for each DRBG. Note that the name of the generalized function call of this section (i.e., **DRBG (...)**) is different than the specific name used for each DRBG (e.g., **Hash_DRBG (...)**).

4. *state_pointer* indicates the internal state to be used and updated during pseudorandom bit generation.

5. *requested_no_of_bits* indicates the number of bits to be returned by the DRBG. If an application always requires the same number of pseudorandom bits to be returned, this parameter **may** be omitted.

6. *requested_strength* is used to request the minimum security strength for the pseudorandom bits to be generated. Note that this parameter is not required for implementations that provide only a single security strength. Note that the *requested_strength* parameter in the DRBG call is a failsafe mechanism. The implementation **shall** check that the value requested is not more than that provided by the instantiation, as determined by the call to the instantiation process (see Section 9.5.1). A call for greater strength **shall** result in an error.

7. Optional *additional_input* may to be provided. This parameter is not required for implementations that will never use *additional_input*.
8. *prediction_resistance_flag* indicates whether or not prediction resistance is to be provided for the pseudorandom bits to be generated (see Section 8.8). This parameter is not required if an implementation will always or never require prediction resistance.
9. *mode* indicates whether pseudorandom bits are requested as part of normal operation or whether testing is being performed.

9.8 Uninstantiate

A process may need to "release" the state space allocated for an instantiation. This may be required, for example, following operational testing of the instantiation process. The **Uninstantiate (...)** process **shall** be performed using the following function call:

status = **Uninstantiate** (*state_pointer*)

where

1. *status* is the indication returned from the uninstantiate process. A status of Success or Failure is expected. The *status* **should** be checked to ensure that the state space has been released.
2. **Uninstantiate (...)** is specified for each DRBG. Note that the name of the generalized function call (i.e., **Uninstantiate (...)**) is different than the specific name used for each DRBG (e.g., **Uninstantiate_Hash_DRBG (...)**).
3. *state_pointer* indicates the internal state to be released.

9.9 Self-Testing of the DRBG

9.9.1 Discussion

A DRBG **shall** perform self testing to obtain assurance that the implementation continues to operate as designed and implemented (operational testing). A DRBG may also be tested to validate that it has been implemented correctly. See Section 11 for a discussion of operational and implementation validation testing.

9.9.2 Specifications

9.9.2.1 Test Specification Variables

Abort_to_error_state (*status_message*)	The abort routine for critical failures that is specified in Section 9.9.2.10.
additional_input_flag	Indicates whether additional input should be provided for testing, where *additional_input_flag* = {*Additional_input_provided*, *No_additional_input_provided*}.
additional_input_text	The text to be used as additional input during the testing of the pseudorandom bit generation and reseeding processes.

Comment [ebb7]: Page: 65
Should there be more than one value ? Should there be different lengths ?

ctr	A count of the number of requests for pseudorandom bits since instantiation or reseeding.
DRBG_specific_parameters	DRBG-specific parameters to be included in the test function calls. Thee parameters are identified for each DRBG in Section 10, if required. Note that the presence of these parameters may require additional steps in the testing process. This will be addressed for each DRBG, when necessary.
*entropy_input*_1, *entropy_input*_2	The entropy input returned from the **Get_entropy (...)** function.
ES_Selftest ( )	The entopy input source testing function specified in Section 9.9.2.9.
expected_instantiated_state_with_personalization_string (*strength, prediction_resistance_flag*)	An array of expected values of the state that is compared against the state generated during instantiation testing when a *personalization_string* is used.
expected_instantiated_state_with_no_personalization_string(*strength, prediction_resistance_flag*)	An array of expected values of the state that is compared against the state generated during instantiation testing when no *personalization_string* is used.
expected_large_string_with_no_prediction_resistance (*strength, additional_input_flag*)	An array of expected values for each strength when a large number of pseudorandom bits is requested from the generation process without prediction resistance.
expected_large_string_with_prediction_resistance (*strength, additional_input_flag*)	An array of expected values for each strength when a large number of pseudorandom bits is requested from the generation process with prediction resistance.
expected_reseeded_state (*strength*)	An array of expected states when reseeding is performed; a state is defined for each *strength* to be tested.
expected_small_string_with_no_prediction_resistance (*strength, additional_input_flag*)	An array of expected values for each strength when a small number of pseudorandom bits is requested from the generation process without prediction resistance.
expected_small_string_with_prediction_resistanc e(*strength, additional_input_flag*)	An array of expected values for each strength when a small number of pseudorandom bits is requested from the generation process with prediction resistance.
Get_entropy (*min_entropy, min_length, max_length, mode*)	

	A function that acquires entropy input from an entropy source. See Section 9.5.2
large_no_of_bits	The number of pseudorandom bits requested during testing of the pseudorandom bit generation process. This value is larger than a block of bits produced by the DRBG and is specific to the DRBG and its specification. See the DRBGs in Section 10 for an appropriate value for a given implementation.
last_state	The index of the last *state* in an implementation.
max_length	The maximum length for a string of bits.
max_strength	The maximum security strength supported by a DRBG implementation (as opposed to a DRBG instantiation).
max_updates	The maximum number of requests for the generation of pseudorandom bits before reseeding is required.
min_entropy	The minimum amount of entropy required.
min_length	The minimum length of a string of bits.
mode	An indication of whether requests for entropy input are for normal operation or for testing. Possible values are *mode* = {*Normal_operation, Fixed_entropy_input_1, Fixed_entropy_input_2,...., Failure*}, where *Fixed_entropy_input_n* selects a fixed value as the entropy input.
Null	A null (i.e., empty) string.
prediction_resistance_flag	Indicates whether or not prediction resistance requests should be handled. Possible values are *prediction_resistance_flag* = {*No_prediction_resistance*, *Allow_prediction_resistance*}.
pseudorandom_bits	The pseudorandom bits that are generated during a single call to the generation process.
requested_strength	The requested strength during a pseudorandom bit generation process.
small_no_of_bits	The number of pseudorandom bits requested during testing of the pseudorandom bit generation process. This value is smaller than a block of bits produced by the DRBG and is specific to the DRBG and its specification. See the DRBGs in Section 10 for an appropriate value for a given implementation.
state (*state_pointer*)	An array of states for for different DRBG instantiations. A state is carried between DRBG calls. The *state* consists of multiple elements that are accessed as *state* (*state_pointer*).*element*. The state elements are specific to each DRBG. The *state* may be considered as *Empty*, *Test_not_empty* or contain the state for an instantiation. *Test_not_empty* **shall** be an illegal value (i.e., not *Empty* and not a recognized normal operational value for the state).

state_pointer	A pointer to the state space for a given DRBG instantiation. An invalid/incorrect state pointer is specified as *Invalid_state_pointer*.
status	The status returned from a function call, where *status* = "Success" or a failure message.
strength	The security strength to be provided by the DRBG instantiation.
temp	A temporary value.
Test_Generation (*strength*, *state_pointer*)	The pseudorandom bit generation testing function specified in Section 9.9.2.4.
Test_Generation_Error_Handling (*strength*, *state_pointer*)	The testing function specified in Section 9.9.2.7 for error handling by the pseudorandom bit generation process.
Test_Instantiation (*strength*, *prediction_resistance_flag*)	The instantiation testing function specified in Section 9.9.2.3.
Test_Instantiation_Error_Handling (*strength*)	The testing function specified in Section 9.9.2.6 for error handling by the instantiation process.
Test_Reseeding (*strength*, *state_pointer*)	The reseeding test function specified in Section 9.9.2.5.
Test_Reseeding_Error_Handling (*state_pointer*)	The testing function specified in Section 9.9.2.8 for error handling by the reseeding process.
Test_personalization_string	A personalization string to be used during testing.
Uninstantiate (*state_pointer*)	The uninstantiate process discussed in Section 9.8 and specified for each DRBG in Section 10.

Comment [ebb8]: Page: 68
Is a single string sufficient ? Should there be different lengths ?

9.9.2.2 Test_DRBG (...)

Test_DRBG (...) shall test each DRBG process that resides in a DRBG boundary. As discussed in Section 8.3, the testing function is contained within the same DRBG boundary as the DRBG process being tested. Therefore, the internal state values are available for modification and examination by the testing function. When an error is detected during DRBG testing, the process **shall** enter an error state (see Section 9.9.2.10).
Each DRBG function within a DRBG boundary **shall** be tested in accordance with Section 11.4 (operational testing) using the following process.
The following **Test_DRBG (...)** process is the highest level routine of the tests. The steps used by an implementation depends on the DRBG processes that are available in the DRBG boundary.

- Steps 1 and 2 **shall** be present if a source of entropy input is available.
- Step 3 **shall** include all security strengths implemented.
- Steps 3.1, 3.2, 3.7, 4 and 5 **shall** be present if the instantation process is available and prediction resistance is not required.
- Steps 3.8, 3.9, 3.14, 4 and 5 **shall** be present if the instantation process is available and prediction resistance can be handled.
- Steps 3.3, 3.4, 6 and 7 **shall** be present if the generation process is available and prediction resistance is not required. Note that if the instantiation process is not available, the *state_pointer* **shall** be set to a state space that is not otherwise used (e.g., reserved for testing only).
- Steps 3.10, 3.11, 6 and 7 **shall** be present if the generation process is available and prediction resistance can be handled. Note that if the instantiation process is not available, the *state_pointer* **shall** be set to a state space that is not otherwise used (e.g., reserved for testing only).
- Steps 3.5, 3.6, 8 and 9 **shall** be present if the reseeding process is available and prediction resistance is not required. Note that if the instantiation process is not available, the *state_pointer* **shall** be set to a state space that is not otherwise used (e.g., reserved for testing only).
- Steps 3.12, 3.13, 8 and 9 **shall** be present if the reseeding process is available and prediction resistance can be handled. Note that if the instantiation process is not available, the *state_pointer* **shall** be set to a state space that is not otherwise used (e.g., reserved for testing only).
- Step 10 **shall** be present for all implementations.

The following process or its equivalent **shall** be used to test a DRBG implementation.

Test_DRBG ( ):

Input: None
Output: string *status*.
Process:

```
1. status = ES_Selftest ( ).                  Comment : Test the entropy input
                                               source. See Section 9.9.2.9.
2. If (status ≠ "Success"), then Abort_to_error_state ("Self testing failure of the
   entropy input source").
                                        Comment : Test normal operation for each
                                        strength supported by a DRBG
                                        implementation.
3. For strength = 80, 112, 128, 192, 256
                                               Comment : Test the instantiation
                                               process with no prediction resistance.
                                               See Section 9.9.2.3.
    3.1 (status, state_pointer) = Test_Instantiation (strength,
        No_prediction_resistance).
    3.2 If (status ≠ "Success"), then Abort_to_error_state ("Self testing failure
        during instantiation (no prediction resistance):" || status).
```

Comment [ebb9]: Page: 69
Have not yet included tests for in between sizes.

```
                                    Comment : Test the generation
                                    process. See Section 9.9.2.4.
    3.3  status = Test_Generation (strength, state_pointer).
    3.4  If (status ≠ "Success"), then Abort_to_error_state ("Self testing failure
         during pseudorandom bit generation (no prediction resistance):" ||
         status).
                                    Comment : Test the reseeding process.
                                    See Section 9.9.2.5.
    3.5  status = Test_Reseeding (strength, state_pointer).
    3.6  If (status ≠ "Success"), then Abort_to_error_state ("Self testing failure
         during reseeding (no prediction resistance) :" || status).
    3.7  status = Uninstantiate (state_pointer).
                                    Comment : Test the instantiation
                                    process with prediction resistance. See
                                    Section 9.9.2.3.
    3.8  (status, state_pointer) = Test_Instantiation (strength,
         Allow_prediction_resistance).
    3.9  If (status ≠ "Success"), then Abort_to_error_state ("Self testing failure
         during instantiation (with prediction resistance):" || status).
                                    Comment : Test the generation
                                    process. See Section 9.9.2.4.
    3.10 status = Test_Generation (strength, state_pointer).
    3.11 If (status ≠ "Success"), then Abort_to_error_state ("Self testing failure
         during pseudorandom bit generation (with prediction resistance):" ||
         status).
                                    Comment : Test the reseeding process.
                                    See Sectuion 9.9.2.5.
    3.12 status = Test_Reseeding (strength, state_pointer).
    3.13 If (status ≠ "Success"), then Abort_to_error_state ("Self testing failure
         during reseeding (with prediction resistance) :" || status).
    3.14 status = Uninstantiate (state_pointer).
                                    Comment : Test error handling. Note
                                    that strength should now be the
                                    highest strength available in an
                                    implementation
                                    Comment : Test error handling during
                                    instantiation. See Section 9.9.2.6.
4. status = Test_Instantiation_Error_Handling (strength).
5. If (status ≠ "Success"), then Abort_to_error_state ("Self testing failure during
   instantiation error handling test :" || status).
```

```
                                        Comment : Test error handling during
                                        pseudorandom bit generation. See
                                        Section 9.9.2.7.
6. status = Test_Generation_Error_Handling (strength, state_pointer).
7. If (status ≠ "Success"), then Abort_to_error_state ("Self testing failure during
   pseudorandom bit generation error handling test :" || status).
                                        Comment : Test error handling during
                                        reseeding. See Section 9.9.2.8.
8. status = Test_Reseeding_Error_Handling (strength, state_pointer).
9. If (status ≠ "Success"), then Abort_to_error_state ("Self testing failure during
   reseeding error handling test :" || status).
10. Return ("Success").
```

9.9.2.3 Test_DRBG_Instantiation (...)

The following **Test_Instantiation (...)** process **shall** be present when the DRBG boundary contains the instantiation process. Calls to **Instantiate_DRBG (...)** **shall** be considered as calls to the instantiation process for the appropriate DRBG (e.g., **Instantiate_Hash_DRBG (...)**).

- Steps 1-3 **shall** be present if an implementation can handle a personalization string.
- Step 4 **shall** be present if steps 5-7 are present.
- Steps 5-7 **shall** be present if an implementation can handle a Null personalization string and does not require prediction resistance.
- Step 8 **shall** be present for all implementations.

Note that steps 5-7 are not followed by a call for uninstantiation. This will allow the final instantiation to be used for subsequent testing (e.g., for pseudorandom bit generation). The test sets may be reordered, but the final test set **shall** provide an instantiation that can be used for further testing.

The following process or its equivalent **shall** be used to test a DRBG instantiation process.

Test_ Instantiation ( ):

```
Input: integer strength, prediction_resistance_flag.
Output: string status, integer state_pointer.
Process:
                                        Comment: Test with a personalization
                                        string. See Section 9.5.1.
  1. (status, state_pointer) = Instantiate_DRBG (strength,
     prediction_resistance_flag, Test_personalization_string,
     DRBG_specific_parameters, Fixed_entropy_input_1).
  2. If (status ≠ "Success"), then Return (status).
  3. If (state (state_pointer) ≠
     expected_instantiated_state_with_personalization_string (strength,
     prediction_resistance_flag), then Return ("Incorrect test state using a
     personalization_string ").
                                        Comment: Remove the state. See
                                        Section 9.8.
```

4. *status* = **Uninstantiate** (*state_pointer*).

   Comment: Test with no personalization string. See Section 9.5.1.

5. (*status, state_pointer*) = **Instantiate_DRBG** (*strength, prediction_resistance_flag*, *Null*, *DRBG_specific_parameters*, *Fixed_entropy_input*_1).
6. If (*status* ≠ "Success"), then **Return** (*status*).
7. If (*state* (*state_pointer*) ≠ *expected_instantiated_state_with_no_personalization_string* (*strength, prediction_resistance_flag*), then **Return** ("Incorrect test state with a null *personalization_string*").
8. **Return** ("Success", *state_pointer*).

9.9.2.4 Test_Generation (...)

The following **Test_Generation (...)** process **shall** be present when the DRBG boundary includes the generation process. Calls to **DRBG (...)** **shall** be considered as calls to the generation process for the appropriate DRBG (e.g., **Hash_DRBG (...)**).

- The appropriate steps of steps 1-12 **shall** be present if a generation process does not require prediction resistance.
  - Steps 1-3 and 7-9 **shall** be present when an implemenation is capable of handling *additional_input.*
  - Steps 4-6 and 10-12 **shall** be present when an implemenation can handle null *additional_input.*
- Step 13 **shall** be present if an implementation does not require prediction resistance at all times.
- The appropriate steps of steps 14-25 **shall** be present if a generation process can handle prediction resistance.
  - Steps 14-16 and 20-22 **shall** be present when an implemenation is capable of handling *additional_input.*
  - Steps 17-19 and 23-25 **shall** be present when an implemenation can handle null *additional_input.*
- Steps 26-28 **shall** be present if an implementation is unable to reseed from the generation process, but **shall** be omitted otherwise.
- Steps 29-32 **shall** be present when reseeding is available online, but **shall** be omitted otherwise.

The following process or its equivalent **shall** be used to test a pseudorandom bit generation process.

Test_Generation ( ):

Input: integer *requested_strength, state_pointer.*
Output: string *status.*
Process:

Comment : Request the generation of a small number of bits with an *additional_input* string and no

```
                                        prediction resistance. See Section
                                        9.7.1.

1.  (status, pseudorandom_bits) = DRBG (state_pointer, small_no_of_bits,
    requested_strength, additional_input_text, No_prediction_resistance,
    Fixed_entropy_input_1).
2.  If (status ≠ "Success"), then Return (status).
3.  If (pseudorandom_bits ≠ expected_small_string_with no prediction_resistance
    (requested_strength, Additional_input_provided)), then Return ("Incorrect bits
    returned when additional_input but no prediction resistance is provided, and a
    small string is requested").
                                        Comment : Request the generation of
                                        a small number of bits with no
                                        additional_input string and no
                                        prediction resistance. See Section
                                        9.7.1.

4.  (status, pseudorandom_bits) = DRBG (state_pointer, small_no_of_bits,
    requested_strength, Null, No_prediction_resistance, Fixed_entropy_input_1).
5.  If (status ≠ "Success"), then Return (status).
6.  If (pseudorandom_bits ≠ expected_small_string_with_no_prediction_resistance
    (rerquested_strength, No_additional_input_provided)), then Return
    ("Incorrect bits returned when no additional_input and no prediction resistance
    is provided, and a small string is requested ").
                                        Comment : Request the generation of
                                        a larger number of bits with an
                                        additional_input string. See Section
                                        9.7.1.

7.  (status, pseudrandom_bits) = DRBG (state_pointer, large_no_of_bits,
    requested_strength, additional_input_text, No_prediction_resistance,
    Fixed_entropy_input_1).
8.  If (status ≠ "Success"), then Return (status).
9.  If (pseudorandom_bits ≠ expected_ large_string_with no_prediction_resistance
    (requested_strength, Additional_input_provided)), then Return ("Incorrect bits
    returned when additional_input but no prediction resistance is provided, and a
    large string is requested").
                                        Comment : Request the generation of
                                        a larger number of bits when no
                                        additional_input is provided. See
                                        Section 9.7.1.

10. (status, pseudrandom_bits) = DRBG (state_pointer, large_no_of_bits,
    requested_strength, Null, No_prediction_resistance, Fixed_entropy_input_1).
11. If (status ≠ "Success"), then Return (status).
12. If (pseudorandom_bits ≠ expected_large_string (requested_strength,
    No_additional_input)), then Return ("Incorrect bits returned when no
```

additional_input and no prediction resistance is provided, and a large string is requested").

Comment : Return if there is no prediction resistance capability in the *state*. See Section 9.7.1.

13. If (*state* (*state_pointer*)*.prediction_resistance_flag*) = *No_prediction_resistance*), then go to step 26.

Comment : Test the prediction_resistance capability.

Comment : Request the generation of a small number of bits with an *additional_input* string. See Section 9.7.1.

14. (*status*, *pseudorandom_bits*) = **DRBG** (*state_pointer*, *small_no_of_bits*, *requested_strength*, *additional_input_text*, *Provide_prediction_resistance*, *Fixed_entropy_input*_2).
15. If (*status* ≠ "Success"), then **Return** (*status*).
16. If (*pseudorandom_bits* ≠ *expected_small_string_with_prediction_resistance* (*requested_strength*, *Additional_input_provided*)), then **Return** ("Incorrect bits returned when *additional_input* and prediction resistance is provided, and a small string is requested").

Comment : Request the generation of a small number of bits with no *additional_input* string. See Section 9.7.1.

17. (*status*, *pseudorandom_bits*) = **DRBG** (*state_pointer*, *small_no_of_bits*, *requested_strength*, *Null*, *Provide_prediction_resistance*, *Fixed_entropy_input*_3).
18. If (*status* ≠ "Success"), then **Return** (*status*).
19. If (*pseudorandom_bits* ≠ *expected_small_string_with prediction_resistance* (*requested_strength*, *No_additional_input_provided*)), then **Return** ("Incorrect bits returned when no *additional_input* is provided but prediction resistance is requested, and a small string is requested ").

Comment : Request the generation of a larger number of bits with an *additional_input* string. See Section 9.7.1.

20. (*status*, *pseudrandom_bits*) = **DRBG** (*state_pointer*, *large_no_of_bits*, *requested_strength*, *additional_input_text*, *Provide_prediction_resistance*, *Fixed_entropy_input*_4).
21. If (*status* ≠ "Success"), then **Return** (*status*).

22. If (*pseudorandom_bits* ≠ *expected_ large_string_with prediction_resistance* (*requested_strength, Additional_input_provided*)), then **Return** ("Incorrect bits returned when *additional_input* is provided, but prediction resistance is requested, and a large string is requested").

    Comment : Request the generation of a larger number of bits when no *additional_input* is provided. See Section 9.7.1.

23. (*status, pseudrandom_bits*) = **DRBG** (*state_pointer, large_no_of_bits,requested_strength*, Null, *Provide_prediction_resistance, Fixed_entropy_input_5*).
24. If (*status* ≠ "Success"), then **Return** (*status*).
25. If (*pseudorandom_bits* ≠ *expected_large_string_with prediction_resistance* (*requested_strength, No_additional_input*)), then **Return** ("Incorrect bits returned when no *additional_input* is provided, but prediction resistance is requested, and a large string is requested").

    Comment : Test the end of the DRBG when reseeding and prediction resiatence is not available (i.e., step 3 of **Hash_DRBG (...)**). See Section 9.7.1

26. *state (state_pointer).ctr = max_updates.*
27. (*status, pseudorandom_bits*) = **DRBG** (*state_pointer, small_no_of_bits, requested_strength, additional_input_text, No_prediction_resistance, Fixed_entropy_input_1*).
28. If (*status* ≠ "DRBG can no longer be used. Please re-instantiate or reseed"), then **Return** ("Incorrect result for *max_updates* test").

    Comment : Test the reseeding capability when ctr ≥ max_updates and the reseeding process is available (i.e., step 12 of **Hash_DRBG (...)**).

29. *state(state_pointer).ctr = max_updates - 1.*
30. (*status, pseudorandom_bits*) = **DRBG** (*state_pointer, small_no_of_bits, requested_strength, additional_input_text, No_prediction_resistance, Fixed_entropy_input_6*).
31. If (*status* ≠ "Success"), then **Return** (*status*).
32. If (*pseudorandom_bits* ≠ *string_after_reseeding* (*requested_strength*)), then **Return** ("Incorrect reseeding process").
33. **Return** ("Success").

9.9.2.5 Test_Reseeding (...)

The following **Test_ Reseeding (...)** process **shall** be available when an implementation has the reseeding process. Calls to **Reseed_DRBG_Instantiation (...)** **shall** be considered

as calls to the reseeding process for the appropriate DRBG (e.g., **Reseed_Hash_DRBG_Instantiation (...)**).
The following process or its equivalent **shall** be used to test a DRBG reseeding process.
Test_Reseeding ( ):

```
Input: integer strength, state_pointer.
Output: string status.
Process:
    1. status = Reseed_DRBG_Instantiation (state_pointer, Fixed_entropy_input_7).
    2. If (status ≠ "Success"), then Return (status).
    3. If (state(state_pointer) ≠ expected_reseeded_state (strength)), then Return ("Incorrect reseed test state").
    4. Return ("Success").
```

9.9.2.6 Test_Instantiation_Error_Handling (...)

The following **Test_Instantiation_Error_Handling (...)** process **shall** be available when an implementation has the instantiation process. Calls to **Instantitate_DRBG (...) shall** be considered as calls to the instantiation process for the appropriate DRBG (e.g., **Instantiate_Hash_DRBG (...)**).

- Note that *strength* **shall** be the highest strength available in an implementation.
- If the *No_prediction_resistance* flag in steps 1, 3 and 6 cannot be handled by an implementation, the flag **shall** be changed to *Allow_prediction_resistance*.
- If the implementation cannot handle a personalization string, then *Test_personalization_string* **shall** be changed to *Null* in steps 1, 3 and 6.

The following process or its equivalent **shall** be used to test error handling by an instantiation process.
Test_Instantiation_Error_Handling ( ) :

Comment [ebb10]: Page: 77
Don't know how to check prediction resistance capability flag failure.

```
Input: integer strength.
Output: string status.
Process:
                                        Comment : Test requested_strength
                                        check failure. The strength ≥ the last
                                        strength tested by Test_DRBG (...).
    1. (status, state_pointer) = Instantiate_DRBG (strength + 1,
       No_prediction_resistance, Test_personalization_string,
       DRBG_specific_parameters, Fixed_entropy_input_1).
    2. If (status = "Success"), then Return ("Accepted incorrect strength").
                                        Comment : Test Get_entropy (...)
                                        status check failure.
    3. (status, state_pointer) = Instantiate_DRBG (strength,
       No_prediction_resistance, Test_personalization_string,
       DRBG_specific_parameters, Failure).
    4. If (status = "Success"), then Return ("Get_entropy failure not detected").
```

```
                                  Comment : Test the
                                  Find_state_space (...) error handling
                                  process. Fill any unused state space.
5. For $i = 0$ to $last\_state$ do
       If ($state\ (i) = Empty$), then $state\ (i) = Test\_not\_empty$.
6. ($status$, $state\_pointer$) = Instantiate_DRBG ($strength$,
   $No\_prediction\_resistance$, $Test\_personalization\_string$,
   $DRBG\_specific\_parameters$, $Fixed\_entropy\_input\_8$).
7. If ($status$ = "Success"), then Return ("Did not detect the full state space").
8. For $i = 0$ to $last\_state$ do
       If ($state\ (i) = Test\_not\_empty$), then $state\ (i) = Empty$.
9. Return ("Success").
```

9.9.2.7 Test_Generation_Error_Handling (...)

The following **Test_Generation_Error_Handling (...)** process **shall** be available when an implementation has the pseudorandom bit generation process. Calls to **DRBG (...) shall** be considered as calls to the generation process for the appropriate DRBG (e.g., **Hash_DRBG (...)**).

- Note that the *requested_strength* is the highest strength available for the implementation.
- If the implementation cannot handle *additional_input_text* or the *No_prediction_resistance* flag, then step 1 **shall** be modified to a call that can be handled (e.g., by changing to the *Allow_prediction_resistance* flag).
- Steps 1 and 2 **shall** be present when the generation process includes a check for an appropriate state pointer.
- Steps 3-7 **shall** be present when the generation process has no ability to automatically reseed.
- Steps 8 and 9 **shall** be present when the generation process checks for an appropriate security strength request.
- Steps 10 and 17 **shall** be present to test prediction resistance.
- Steps 11-13 **shall** be present when prediction resistance is supported, and the generation process checks whether a prediction resistance capability was instantiated.
- Steps 14-16 **shall** be present when both reseeding and prediction resistance are supported.
- Steps 18-20 **shall** be present when automatic reseeding is available and a check is made to determine if *max_updates* has been reached.
- Step 21 **shall** always be included.

The following process or its equivalent **shall** be used to test error handling by a pseudorandom bit generation process.

Test_Generation_Error_Handling ( ) :

Input: integer *requested_strength, state_pointer*.

Output: string *status*.
Process:

```
                                        Comment : Test state_pointer
                                        checking.
1.  (status, entropy_input) = DRBG (Invalid_state_pointer, small_number_of_bits,
    requested_strength, additional_input_text, No_prediction_resistance,
    Fixed_entropy_input_1).
2.  If (status = "Success"), then Return ("Accepted incorrect state_pointer").
                                        Comment : Test abort when
                                        max_updates is reached and reseeding
                                        is unavailable.
3.  temp = state (state_pointer).ctr.
4.  state (state_pointer).ctr = max_updates.
5.  (status, entropy_input) = DRBG (state_pointer, small_no_of_bits,
    requested_strength, additional_input_text, No_prediction_resistance,
    Fixed_entropy_input_1).
6.  If (status = "Success"), then Return ("Incorrect operation when ctr =
    max_updates").
7.  state (state_pointer).ctr = temp.
                                        Comment : Test requested_strength
                                        checking.
8.  (status, entropy_input) = DRBG (state_pointer, small_no_of_bits,
    requested_strength + 1, additional_input_text, No_prediction_resistance,
    Fixed_no_of_bits_1).
9.  If (status = "Success"), then Return ("Accepted incorrect
    requested_strength").
                                        Comment : Test inappropriate
                                        prediction_resistance_request
                                        checking.
10. temp = state (state_pointer).prediction_resistance_flag.
11. state (state_pointer).prediction_resistance_flag = No_prediction_resistance.
12. (status, entropy_input) = DRBG (state_pointer, small_no_of_bits,
    requested_strength, additional_input_text, Provide_prediction_resistance,
    Fixed_no_of_bits_2).
13. If (status = "Success"), then Return ("Incorrect handling of prediction
    resistance request").
                                        Comment : Test reseeding error when
                                        prediction resistance requested.
14. state (state_pointer).prediction_resistance_flag =
    Provide_prediction_resistance.
15. (status, entropy_input) = DRBG (state_pointer, small_no_of_bits,
    requested_strength, additional_input_text, Provide_prediction_resistance,
    Failure).
```

```
16. If (status = "Success"), then Return ("Failure indication from reseed request
    when prediction resistance requested").
17. state (state_pointer).prediction_resistance_flag = temp.
                                        Comment : Test reseeding when ctr
                                        reaches max_updates.
18. state (state_pointer).ctr = max_updates - 1.
19. (status, entropy_input) = DRBG (state_pointer, small_no_of_bits,
    requested_strength + 1, additional_input_text, Provide_prediction_resistance,
    Failure).
20. If (status = "Success"), then Return ("Incorrect reseed handling when ctr ≥
    max_updates").
21. Return ("Success").
```

9.9.2.8 Test_Reseeding_Error_Handling (...)

The following **Test_DRBG_Reseeding_Error_Handling (...)** process **shall** be available when an implementation has the reseeding process. Calls to **Reseed_Instantiation (...)** **shall** be considered as calls to the reseeding process for the appropriate DRBG (e.g., **Reseed_Hash_DRBG_Instantation (...)**).

- Steps 3 and 4 **shall** be present if entropy can be readily obtained.

The following process or its equivalent **shall** be used to test error handling by a reseeding process.

Test_Reseeding_Error_Handling ( ) :

```
Input: integer state_pointer.
Output: string status.
Process:
                                        Comment : Test state_pointers check
                                        failure.
    1. status = Reseed_Instantiation (Invalid_state_pointer, Fixed_entropy_input_2).
    2. If (status = "Success"), then Return ("Accepted incorrect state_pointer").
                                        Comment : Test Get_entropy (...)
                                        status check failure.
    3. status = Reseed_ Instantiation (state_pointer, Failure).
    4. If (status = "Success"), then Return ("Get_entropy failure not detected").
                                        Comment : Test check of old and new
                                        entropy_input.
    5. state (state_pointer).transformed_seed = Fixed_entropy_input_2.
    6. status = Reseed_ Instantiation (state_pointer, Fixed_entropy_input_2).
    7. If (status = "Success"), then Return ("Entropy input failure not detected").
    8. Return ("Success").
```

9.9.2.9 ES_Selftest (...)

The concept of an entropy input source selftest is introduced in Part 1 of this Standard. This test **shall** consist of the following steps. Let *max_strength* be the maximum strength to be supported by the DRBG implementation; let *min_length* be the appropriate minimum length of the entropy input for the DRBG when it supports the maximum strength; and let *max_length* be the maximum length of the entropy input for the DRBG when it supports the maximum strength.

The following process or its equivalent **shall** be used to test the entropy input source.

ES_Selftest (...):

```
    Input: None..
    Output: string status.
    Process:
                                        Comment: Obtain two strings.
        1. min_entropy = max (128, max_strength).
        2. (status, entropy_input_1) = Get_entropy (min_entropy, min_length,
           max_length, Normal_operation).
        3. If (status ≠ "Success"), then Return ("Failure indication returned by the
           Get_entropy source").
        4. (status, entropy_input_2) = Get_entropy (min_entropy, min_length,
           max_length, Normal_operation).
        5. If (status ≠ "Success"), then Return ("Failure indication returned by the
           Get_entropy source").
                                        Comment : Compare the two strings.
        6. If (len (entropy_input_1) ≠ len (entropy_input_2)), then Return ("Success").
        7. If (entropy_input_1 = entropy_input_2), then Return ("Entropy input source
           failure").
        8. Return ("Success").
```

9.9.2.10 Abort_to_error_state (...)

Critical errors, such as the failure of the entopy input source, **shall** call the **Abort_to_error_state (...)** process specified below.

The following or an equivalent process **shall** be used as the **Abort_to_error_state (...)** function:

Abort_to_error_state (...):

```
    Input: string status.
    Output: None.
    Process:
        1. Display ("status").          Comment : Display the error
                                        indication message.
        2. For i = 0 to last_state      Comment: Uninstantiate all states.
               Uninstantiate (i).
        3. Abort ().                    Comment: Abort the DRBG.
```

9.10 Error Handling

The expected errors are indicated for each DRBG in Section 10 and for testing in Section 9.9. The error handling routine **shall** 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 (see Section 9.9.2.9).

Most errors during normal operation are 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 organization security policy. For example, if a failure of "Invalid *requested_strength*" is returned, a security strength higher than the DRBG can support has been requested. The user **may** reduce the requested strength if the organization's security policy allows the information to be protected using a lower security strength, or the user **shall** use another appropriately instantiated DRBG for the usage class.

Failures that indicate that the entropy source has failed or that the DRBG failed operational testing (see Sections 9.9.2.9 and 11.4) **shall** be perceived as complete DRBG failures. The indicated DRBG problem **shall** be corrected before re-instantiating the DRBG and requesting pseudorandom bits.