10.2 DRBGs Based on Block Ciphers

10.2.1 Discussion

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

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

Table 3 specifies the security strengths and entropy and seed requirements that **shall** be used for each Approved block cipher algorithm.

Table 3: Security Strengths, Entropy and Seed Length Requirements for Approved Block Cipher Algorithms

Block Cipher Algorithm	Security Strengths	Required Minimum Entropy	Entropy Input Lengths (in bits)	Seed Length (in bits)
2 key TDEA	80	128	$128\text{-}2^{35}$	176
3 key TDEA	80, 112	128	$128\text{-}2^{35}$	232
AES-128	80, 112, 128	128	$128\text{-}2^{35}$	256
AES-192	80, 112, 128, 192	192	$192\text{-}2^{35}$	320
AES-256	80, 112, 128, 192, 256	256	$256\text{-}2^{35}$	384

10.2.2 CTR_DRBG

10.2.2.1 Discussion

CTR_DRBG (...) uses an Approved block cipher algorithm in the counter mode as specified in [SP 800-38A]. The same block cipher algorithm and key length **shall** be used for all block cipher operations. The block cipher algorithm and key size **shall** meet or exceed the security requirements of the consuming application. Table 3 in Section 10.2.1 specifies the entropy and seed length requirements that **shall** be used for each block cipher algorithm to meet the required security level.

Figure 12 depicts the **CTR_DRBG (...)**. **{Note : Figure to be inserted later.}**

10.2.2.2 Interaction with CTR_DRBG

10.2.2.2.1 Instantiating CTR_DRBG

Prior to the first request for pseudorandom bits, the **CTR_DRBG (...) shall** be instantiated using the following call:

(*status, state_handle*) = **Instantiate_CTR_DRBG** (*requested_strength, prediction_resistance_flag, personalization_string*)

as described in Sections 9.5.1 and 10.2.2.3.4.

10.2.2.2.2 Reseeding CTR_DRBG

When a **CTR_DRBG (...)** instantiation requires reseeding, the DRBG **shall** be reseeded using the following call:

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

as described in Sections 9.6.2 and 10.2.2.3.5.

10.2.2.2.3 Generating Pseudorandom Bits Using CTR_DRBG

An application may request the generation of pseudorandom bits by **CTR_DRBG (...)** using the following call:

(*status, pseudorandom_bits*) = **CTR_DRBG** (*state_handle, requested_no_of_bits, requested_strength, additional_input, prediction_resistance_request_flag*)

as discussed in Sections 9.7.2 and 10.2.2.3.6.

10.2.2.2.4 Removing a CTR_DRBG Instantiation

An application may request the removal of an **CTR_DRBG (...)** instantiation using the following call:

status = **Uninstantiate_CTR_DRBG** (*state_handle*)

as described in Sections 9.8 and 10.2.2.3.7.

10.2.2.2.5 Self Testing of the CTR_)DRBG Process

A **CTR_DRBG (...)** implementation is tested at power-up and on demand using the following call:

status = **Self_Test_CTR_DRBG** ( )

as described in Sections 9.9 and 10.2.2.3.8.

10.2.2.3 Specifications

10.2.2.3.1 General

The instantiation and reseeding of **CTR_DRBG (...)** consists of obtaining a *seed* with the appropriate amount of entropy. The entropy input is used to derive a *seed*, which is then used to derive elements of the initial *state* of the DRBG. The *state* consists of:

1. The value *V*, which is updated each time another *outlen* bits of output are produced (where *outlen* is the number of output bits from the underlying block cipher algorithm).
2. The *Key*, which is updated whenever a predetermined number of output blocks are generated.
3. The key length (*keylen*) to be used by the block cipher algorithm.

4. The security *strength* of the DRBG instantiation.
5. A counter (*reseed_counter*) that indicates the number of requests for pseudorandom bits since instantiation or reseeding.
6. A *prediction_resistance_flag* that indicates whether or not a prediction resistance capability is required for the DRBG.

10.2.2.3.2 CTR_DRBG Variables

The variables used in the description of **KHF_DRBG (...)** are:

additional_input	Optional additional input, which must be ≤ *max_length* bits in length.
Block_Cipher (*Key, V*)	The block cipher algorithm, where *Key* is the key to be used, and *V* is the input block.
Block_Cipher_df (*a, b*)	The block cipher derivation function specified in Section 9.5.4.3. **{Note: The Block_Cipher_df will be specified later.}**
blocklen	The length of the block cipher algorithm's output block.
entropy_input	The bits containing entropy that are used to determine the *seed_material* and generate a *seed*.
Find_state_space ( )	A function that finds an unused *state* in the state space. See Section 9.5.3.
Get_entropy (*min_entropy, min_entropy, max_length*)	A function that acquires a string of bits from an entropy input source. See Section 9.5.2.
Invalid_state_handle	An illegal value for the *state_handle*.
*K*ey	The key used to generate pseudorandom bits.
keylen	The length of the key for the block cipher algorithm.
len (*x*)	A function that returns the number of bits in input string *x*.
max_length	The maximum length of a string for obtaining entropy. When a derivation function is used, this value is implementation dependent, but **shall** be ≤ 2^{35} bits. When a derivation function is **not** used, then *max_length = seedlen*.
max_no_of_states	The maximum number of states and instantiations that an implementation can handle.
max_request_length	The maximum number of pseudorandom bits that may be requested during a single request; this value is implementation dependent, but **shall** be ≤ 2^{35} bits for AES, and ≤ 2^{19} bits for TDEA.

min_entropy The minimum amount of entropy to be obtained from the *entropy_input* source and provided in the *seed*.

Null The null (i.e., empty) string.

old_transformed_entropy_input

The *transformed_entropy_input* from the previous acquisition of *entropy_input* (e.g., used during reseeding).

personalization_string A personalization string of no more than *seedlen* bits (see Section 8.7.1).

prediction_resistance_flag

Indicates whether or not prediction resistance requests should be handled; *prediction_resistance_flag* = {*Allow_prediction_resistance, No_prediction_resistance*}.

prediction_resistance_request_flag

Indicates whether or not prediction resistance is required during a request for pseudorandom bits; *prediction_resistance_request_flag* = {*Provide_prediction_resistance, No_prediction_resistance*}.

pseudorandom_bits The pseudorandom bits produced during a single call to the **KHF_DRBG (...)** process.

requested_no_of_bits The number of pseudorandom bits to be generated.

requested_strength The security *strength* to be provided for the pseudorandom bits to be obtained from the DRBG.

reseed_counter A counter that records the number of times pseudorandom bits were requested since the DRBG instantiation was seeded or reseeded.

reseed_interval The maximum number of requests for the generation of pseudorandom bits before reseeding is required. The maximum value **shall** be $\leq 2^{32}$ for AES, and $\leq 2^{16}$ for TDEA.

seedlen The length of the seed, where *seedlen* = *blocklen* + *keylen*.

seed_material The data used as the *seed*.

state (*state_handle*) An array of states for different DRBG instantiations. A *state* is carried between DRBG calls. For the **CTR_DRBG (...)**, the *state* for an instantiation is defined as *state* (*state_handle*) = {*V, Key, keylen, strength, reseed_counter, prediction_resistance_flag*}. A particular element of the *state* is specified as *state*(*state_handle*).*element*, e.g., *state* (*state_handle*).*V*.

state_handle A pointer to the state space for the given instantiation.

status	The status returned from a function call, where *status* = "Success" or a failure message.
strength	The security strength provided by the DRBG instantiation.
temp	A temporary value.
V	A value in the *state* that is updated whenever pseudorandom bits are generated.

10.2.2.3.3 Internal Function: The Update Function

The **Update (...)** function updates the internal state of the **CTR_DRBG (...)** using *seed_material*, which must be *seedlen* bits in length. The following or an equivalent process **shall** be used as the **Update (...)** function.

Update (...):

Input: string (*seed_material*, *keylen*, *Key*, *V*).

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

Process:

1. *seedlen* = *blocklen* + *keylen*.
2. *temp* = *Null*.
3. While (**len** (*temp*) < *seedlen*) do
   3.1 $V = (V + 1) \bmod 2^{blocklen}$.
   3.2 *output_block* = **Block_Cipher** (*Key*, *V*).
   3.3 *temp* = *temp* || *ouput_block*.
4. *temp* = Leftmost *seedlen* bits of *temp*.
5 *temp* = *temp* ⊕ *seed_material*.
6. *Key* = Leftmost *keylen* bits of *temp*.
7. *V* = Rightmost *blocklen* bits of *temp*.
8. **Return** (*Key*, *V*).

10.2.2.3.4 Instantiation of CTR_DRBG (...)

The following process or its equivalent **shall** be used to initially instantiate the **CTR_DRBG (...)** process.

Instantiate_CTR_DRBG (...):

Input: integer (*requested_strength*, *prediction_resistance_flag*, *personalization_string*).

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

Process:

1. Comment: If TDEA is used.

If (*requested_strength* > 112) then **Return** ("Invalid *requested_strength*", *Invalid_state_handle*).

Comment: If AES is used.

If (*requested_strength* > 256) then **Return** ("Invalid *requested_strength*", *Invalid_state_handle*).

2. If (*prediction_resistance_flag = Allow_prediction_resistance*) and prediction resistance cannot be supported, then **Return** ("Cannot support prediction resistance", *Invalid_state_handle*).

Comment: Set the *strength* to one of the five security strengths, and determine the key length.

3. Comment: If TDEA is the block cipher algorithm.

If (*requested_strength* ≤ 80), then (*strength* = 80; *keylen* = 112)

Else if (*requested_strength* ≤ 112), then (*strength* = 112; *keylen* = 168).

Comment: If AES is the block cipher algorithm.

If (*requested_strength* ≤ 80), then (*strength* = 80; *keylen* = 128)

Else if (*requested_strength* ≤ 112), then (*strength* = 112; *keylen* = 128)

Else (*requested_strength* ≤ 128), then (*strength* = 128; *keylen* = 128)

Else (*requested_strength* ≤ 192), then (*strength* = 192; *keylen* = 192)

Else (*strength* = 256; *keylen* = 256).

4. *seedlen* = *blocklen* + *keylen*. Comment: determine the *seed* length.

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

6. If (*temp* > *max_length*), then **Return (**"*personalization_string* too long", *Invalid_state_handle*)

7. Comment: If a derivation function is available (a source of full entropy may or may not be available).

   7.1 *min_entropy* = *strength* + 64.

   7.2 (*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_entropy*, *max_length*).

   7.3 If (*status* ≠ "Success"), then **Return** ("Failure indication returned by the entropy source" || *status*, *Invalid_state_handle*).

   7.4 *seed_material* = *entropy_input* || *personalization_string*.

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

Comment: If a full entropy source is known to be available and a derivation function is not to be used.

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

7.2 If (*status* ≠ "Success"), then **Return** ("Failure indication returned by the entropy source" || *status*, *Invalid_state_handle*).

Comment: Pad with zeros if the personalization string is too short.

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

7.4 *seed_material* = *entropy_input* ⊕ *personalization_string*.

Comment: Find space in the state table.

8. (*status*, *state_handle*) = **Find_state_space** ( ).
9. If (*status* ≠ "Success"), then **Return** ("No available state space" || *status*, *Invalid_state_pointer*).
10. *Key* = 0. Comment: *keylen* bits.
11. *V* = 0. Comment: *blocklen* bits.
12. (*Key*, *V*) = **Update** (*seed_material*, *keylen*, *Key*, *V*).
13. *reseed_counter* = 0.
14. *state* (*state_handle*) = {*V*, *Key*, *keylen*, *strength*, *reseed_counter*, *prediction_resistance_flag*}.
15. **Return** ("Success", *state_handle*).

Steps 1 and 3 must be implemented to handle the algorithm that is available.

The choice of code at step 7 must be selected based on whether the DRBG will be instantiated with a full-entropy source and whether a derivation function will be used.

If no *personalization_string* will ever be provided, then the *personalization_string* input parameter and steps 5 and 6 be omitted. If a derivation function is available, then step 7.4 may be omitted, and step 7.5 becomes:

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

If full entropy is known to be available and a derivation function is not available, then steps 7.3 and 7.4 are omitted, and step 7.1 becomes:

(status, seed_material) **= Get_entropy** (*seedlen, seedlen, seedlen*)..

If an implementation does not need the *prediction_resistance_flag* as a calling parameter (i.e., the **CTR_DRBG (....)** routine in Section 10.2.2.3.6 either always or never acquires new entropy in step 9), then the *prediction_resistance_flag* in the calling parameters and in the *state* (see step 14) may be omitted, as well as omitting step 2.

10.2.2.3.5 Reseeding a CTR_DRBG (...) Process

The following or an equivalent process **shall** be used to explicitly reseed the **CTR_DRBG (...)** process.

Reseed_CTR_DRBG_Instantiation (...):

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

Output: string *status*.

Process:

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

   Comment: Get the appropriate *state* values.

2. *V* = *state*(*state_handle*).*V*, *Key* = *state*(*state_handle*).*Key*, *keylen* = *state*(*state_handle*).*keylen*, *strength* = *state*(*state_handle*).*strength*, *prediction_resistance_flag* = *state*(*state_handle*).*prediction_resistance_flag*..
3. *seedlen* = *blocklen* + *keylen*.
4. *temp* = **len** (*additional_input*).
5. If (*temp* > *max_length*), then **Return** ("*additional_input* too long").
6. Comment: If a derivation function is available (a source of full entropy may or may not be available.

   6.1 *min_entropy* = *strength* + 64.

   6.2 (*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_entropy*, *max_length*).

   6.3 If (*status* ≠ "Success"), then **Return** ("Failure indication returned by the entropy source" || *status*, *Invalid_state_handle*).

   6.4 *seed_material* = *entropy_input* || *additional_input.*

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

   Comment: If a full entropy source is known to be available and a derivation function is not to be used.

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

   6.2 If (*status* ≠ "Success"), then **Return** ("Failure indication returned by the entropy source" || *status*).

Comment: Pad with zeros if the *additional_input_string* is too short.

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

6.4 *seed_material* = *entropy_input* ⊕ *additional_input*.

7. (*Key*, *V*) = **Update** (*seed_material*, *keylen*, *Key*, *V*).
8. *reseed_counter* = 0.
9. *state*(*state_handle*) = {*V*, *Key*, *keylen*, *strength*, *reseed_counter*, *prediction_resistance_flag* }.
10. **Return** ("Success").

The choice of code at step 6 must be selected based on whether the DRBG will be instantiated with a full-entropy source and whether a derivation function will be used.

If an implementation does not handle *additional_input*, then the *additional_input* parameter of the input may be omitted as well as steps 4 and 5. If a derivation function is available, then step 6.4 may be omitted, and step 6.5 may be changed to:

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

If full entropy is known to be available and a derivation function is not available, then steps 6.3 and 6.4 may be omitted, and step 6.1 may be changed to:

(*status*, *seed_material*) = **Get_entropy** (*seedlen*, *seedlen*, *seedlen*).

10.2.2.3.6 Generating Pseudorandom Bits Using CTR_DRBG (...)

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

CTR_DRBG(...):

Input: integer (*state_handle*, *requested_no_of_bits*, *requested_strength*, *additional_input*, *prediction_resistance_request_flag*).

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

Process:

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

   Comment: Get the appropriate *state* values.

2. *V* = *state*(*state_handle*).*V*, *Key* = *state*(*state_handle*).*Key*, *keylen* = *state*(*state_handle*).*keylen*, *strength* = *state*(*state_handle*).*strength*, *reseed_counter* = *state(state_handle).reseed_counter*, *prediction_resistance_flag* = *state(state_handle).prediction_resistance_flag*.
3. If (*requested_strength* > *strength*), then **Return** ("Invalid *requested_strength*", *Null*).

4. *seedlen* = *blocklen* + *keylen*.
5. *temp* = **len** (*additional_input*).
6. If (*temp* > *max_length*), then **Return** ("additional_input too long", Null).
7. If (*requested_no_of_bits* > *max_request_length*), then **Return** ("Too many bits requested", *Null*).
8. If ((*prediction_resistance_request_flag* = *Provide_prediction_resistance*) and (*prediction_resistance_flag* = *No_prediction_resistance*)), then **Return** ("Prediction resistance capability not instantiated", *Null*).
9. If ((*reseed_counter* ≥ *reseed_interval*) OR (*prediction_resistance_request_flag* = *Provide_prediction_resistance*)), then

   Comment: If reseeding is not available.

   Return ("DRBG can no longer be used. Please re-instantiate or reseed.", *Null*).

   Comment: If reseeding is readily available.

   9.1 *status* = **Reseed_CTR_DRBG** (*state_handle*, *additional_input*).

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

   9.3 *V* = *state*(*state_handle*).*V*, *Key* = *state*(*state_handle*).*Key*, *reseed_counter* = *state(state_handle).reseed_counter*.

   9.4 Go to step 11.

   Comment: When reseeding or prediction resistance is not required.

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

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

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

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

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

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

11. *temp* = *Null*.
12. While (**len** (*temp*) < *requested_no_of_bits*) do:
    12.1 $V = (V + 1) \bmod 2^{blocklen}$.
    12.2 *output_block* = **Block_Cipher** (*Key, V*).
    12.3 *temp* = *temp* || *ouput_block*.
13. *pseudorandom_bits* = Leftmost (*requested_no_of_bits*) of *temp*.

Comment: Update for backtracking resistance.

14. $zeros = 0^{seedlen}$.

Comment: Produce a string of *seedlen* zeros.

15. (*Key, V*) = **Update** (*zeros, keylen, Key, V*)
16. *reseed_counter* = *reseed_counter* + 1.
17. *state*(*state_handle*) = {*V*, *Keykeylen, strength, reseed_counter*, *prediction_resistance_flag*,).
18. **Return** ("Success", *pseudorandom_bits*).

If an implementation will never provide *additional_input*, then the *additional_input* input parameter, steps 5, 6 and 10 can be omitted, and a *Null* string replaces the *additional_input* in step 9.1. If *max_length* ≤ *seedlen*, then step 10.1 may be omitted (i.e., the block cipher derivation function is not required).

If an implementation does not need the *prediction_resistance_flag*, then the *prediction_resistance_flag* may be omitted as an input parameter, and step 8 may be omitted.

If an implementation does not have a reseeding capability, then steps 9.1-9.3 may be omitted, and step 9 becomes:

> If (*reseed_counter* ≥ *reseed*_interval), then **Return** ("DRBG can no longer be used. Please re-instantiate or reseed.", *Null*).

10.2.2.3.7 Removing a CTR_DRBG (...) Instantiation

The following or an equivalent process **shall** be used to remove a **CTR_DRBG (...)** instantiation:

Uninstantiate_CTR_DRBG (...):

Input: integer *state_handle*.

Output: string *status*.

Process:

1. If (*state_handle* > *max_no_of_states*), then **Return** ("Invalid *state_handle*").
2. *state*(*state_handle*) = {*Null, Null,* 0,0, 0, 0}.
3. **Return** ("Success").

10.2.2.3.8 Self Testing of the CTR_DRBG (...)

[Tp be determined]

10.2.3 OFB_DRBG (...)

10.2.3.1 Discussion

OFB_DRBG (...) uses an Approved block cipher algorithm in the output feedback mode as specified in [SP 800-38A]. The same block cipher algorithm and key length **shall** be used for all block cipher operations. The block cipher algorithm and key size **shall** meet or exceed the security requirements of the consuming application. Table 3 in Section 10.2.1 specifies the entropy and seed length requirements that **shall** be used for each block cipher algorithm to meet the required security level.

Figure 13 depicts the **CTR_DRBG (...)**. **{Note : To be inserted later.}**

10.2.3.2 Interaction with OFB_DRBG (...)

10.2.3.2.1 Instantiating OFB_DRBG (...)

Prior to the first request for pseudorandom bits, the **OFB_DRBG (...) shall** be instantiated using the following call:

(*status*, *state_handle*) = **Instantiate_OFB_DRBG** (*requested_strength, prediction_resistance_flag, personalization_string*)

as described in Sections 9.5.1 and 10.2.3.3.4.

10.2.3.2.2 Reseeding an OFB_DRBG (...) Instantiation

When an **OFB_DRBG (...)** instantiation requires reseeding, the DRBG **shall** be reseeded using the following call:

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

as described in Sections 9.6.2 and 10.2.3.3.5.

10.2.3.2.3 Generating Pseudorandom Bits Using OFB_DRBG (...)

An application may request the generation of pseudorandom bits by **OFB_DRBG (...)** using the following call:

(*status*, *pseudorandom_bits*) = **OFB_DRBG** (*state_handle, requested_no_of_bits, requested_strength, additional_input*, *prediction_resistance_request_flag*)

as discussed in Sections 9.7.2 and 10.2.3.3.6.

10.2.3.2.4 Removing an OFB_DRBG (...) Instantiation

An application may request the removal of an **OFB_DRBG (...)** instantiation using the following call:

status = **Uninstantiate_OFB_DRBG** (*state_handle*)

as described in Sections 9.8 and 10.2.3.3.7.

10.2.3.2.5 Self Testing of the OFB_DRBG (...) Process

A **OFB_DRBG (...)** implementation is tested at power-up and on demand using the following call:

status = **Self_Test_OFB_DRBG** ( )

as described in Sections 9.9 and 10.2.3.3.8.

10.2.3.3 Specifications

10.2.3.3.1 General

The instantiation and reseeding of **OFB_DRBG (...)** consists of obtaining a *seed* with the appropriate amount of entropy. The entropy input is used to derive a *seed*, which is then used to derive elements of the initial *state* of the DRBG. The *state* consists of:

1. The value *V*, which is updated each time another *outlen* bits of output are produced (where *outlen* is the number of output bits from the underlying block cipher algorithm).
2. The *Key*, which is updated whenever a predetermined number of output blocks are generated.
3. The key length (*keylen*) to be used by the block cipher algorithm.
4. The security *strength* of the DRBG instantiation.
5. A counter (*reseed_counter*) that indicates the number of requests for pseudorandom bits since instantiation or reseeding.
6. A *prediction_resistance_flag* that indicates whether or not a prediction resistance capability is required for the DRBG.

10.2.3.3.2 OFB_DRBG (...) Variables

The variables for **OFB_DRBG (...)** are the same as those used for the **CTR_DRBG (...)** specified in Section 10.2.2.3.2.

10.2.3.3.3 Internal Function: The Update Function

The **Update (...)** function updates the internal state of the **CTR_DRBG (...)** using *seed_material*, which must be *seedlen* bits in length. The following or an equivalent process **shall** be used as the **Update (...)** function.

Update (...):

Input: string (*seed_material, keylen, Key, V*).

Output: string (*Key, V*).

Process:

1. *seedlen* = *blocklen* + *keylen*.
2. *temp* = *Null*.
3. While (**len** (*temp*) < *seedlen*) do
   3.1 *V* = **Block_Cipher** (*Key*, *V*).
   3.2 *temp* = *temp* || *V*.
4. *temp* = Leftmost *seedlen* bits of *temp*.

5 *temp* = *temp* ⊕ *seed_material*.

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

7. *V* = Rightmost *blocklen* bits of *temp*.

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

Note that the only difference between the update function for **OFB_DRBG (...)** and **CTR_DRBG (..)** is in step 3.

10.2.3.3.4 Instantiation of OFB_DRBG (...)

This process is the same as the instantiation process for **CTR_DRBG (...)** in Section 10.2.2.3.4.

10.2.3.3.5 Reseeding an OFB_DRBG (...) Instantiation

This process is the same as the reseeding process for **CTR_DRBG (...)** in Section 10.2.2.3.5.

10.2.3.3.6 Generating Pseudorandom Bits Using OFB_DRBG (...)

This process is the same as the generation process for **CTR_DRBG (...)** in Section 10.2.2.3.6, except that step 11 **shall** be as follows :

9. While (**len** (*temp*) < *requested_no_of_bits*) do:

   11.1 *V* = **Block_Cipher** (*Key*, *V*).

   11.2 *temp* = *temp* || *V*.

10.2.3.3.7 Removing an OFB_DRBG (...) Instantiation

This process is the same as the uninstantiation process for **CTR_DRBG (...)** in Section 10.2.2.3.7.

10.2.3.3.8 Self Testing of the OFB_DRBG (...)

This is the same as the self testing of **CTR_DRBG (...)** in Section 10.2.2.3.8.

Appendix E : DRBG Selection

E.3 DRBGs Based on Block Ciphers

E.3.1 The Two Constructions: CTR and OFB

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

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

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

E.3.2 Choosing a Block Cipher

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

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

1. For AES-128, there are a maximum of 2^{28} blocks (i.e., 2^{32} bytes = 2^{35} bits) generated per **Generate (...)** request, 2^{32} total **Generate (...)** requests allowed, 2^{128} possible keys, and 2^{128} possible starting blocks.

    a. The probability of an internal collision in a single **Generate (...)** request is never higher than about 2^{-96}, and so the probability of an internal collision in any given **Generate (...)** request is never higher than about 2^{-64}. (This applies only to the OFB-mode, but a collision of this kind would result in a very easy distinguisher.)

    b. The expected probability of an internal collision in a sequence of 2^{28} random 128-bit blocks is about 2^{-74}. Thus, the probability of seeing an internal collision in any of the **Generate (...)** sequences is about 2^{-42}. This probability is low enough that it does not provide an efficient way to distinguish between DRBG outputs and ideal random outputs.

    c. The probability of a key colliding between any two **Generate (...)** requests in the sequence of 2^{32} such requests is never larger than about 2^{-65}. This is also negligible. (For AES-192 and AES-256, this probability is even smaller.)

2. For Two-key TDEA with 112-bit keys and 64-bit blocks, things are a bit different: There are 2^{16} **Generate (...)** requests allowed, and a maximum of 2^{13} blocks (i.e., 2^{16} bytes = 2^{19} bits) generated per **Generate (...)** request. (Note that this breaks the more general model in this document of assuming 2^{64} innocent operations.) In this case:

    a. The probability of an internal collision is never higher than about 2^{-51} per **Generate (...)** request, and with only 2^{16} such requests allowed, the probability of ever seeing such an internal collision in a sequence of requests is never more than about 2^{-35}. (Note that if more requests are allowed, as required by the 2^{64} bound assumed elsewhere in the document, there would be an unacceptably high probability of this event happening at least once.)

    b. The expected probability of an internal collision in a sequence of 2^{13} 64-bit blocks is about 2^{-38}. Thus, the probability of ever seeing an internal collision in 2^{16} output sequences is still an acceptably low 2^{-22}. (Note that if more **Generate (...)** requests are allowed, there would be an unacceptably high probability of this happening, leading to an efficient distinguisher between this DRBG's outputs and ideal random outputs.

    c. The probability of a key colliding between any two of the 2^{16} **Generate (...)** requests is about 2^{-56}, which is negligible. (Note that the probability would be much higher if the number of allowed **Generate (...)** requests is not limited.)

To summarize: block size matters much more than the choice of DRBG construction that is used. The limits on the numbers of **Generate (...)** requests and the number of output bits per request require frequent reseeding of the DRBG. Furthermore, the limits

guarantee that even with reseeding, an attacker that is given a really long sequence of DRBG outputs from several reseedings cannot distinguish that output sequence from random reliably. The block cipher DRBGs used with TDEA are suitable for low-throughput applications, but not for applications requiring really large numbers of DRBG outputs. **For concreteness, if an application is going to require more than 2^{32} output bytes (2^{35} bits) in its lifetime, that application should not use a block cipher DRBG with TDEA or any other 64-bit block cipher.**

E.3.3 E.3.3 Conditioned Entropy Sources and the Derivation Function

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