10.3 Deterministic RBGs Based on Number Theoretic Problems

10.3.1 Discussion

A DRBG can be designed to take advantage of number theoretic problems (e.g., the discrete logarithm problem). If done correctly, such a generator's properties of randomness and/or unpredictability will be assured by the difficulty of finding a solution to that problem. Section 10.3.2 specifies a DRBG based on elliptic curves; Section 10.3.3 specifies a DRBG based on the RSA integer factorization problem.

10.3.2 Dual Elliptic Curve Deterministic RBG (Dual_EC_DRBG)

10.3.2.1 Discussion

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

Dual_EC_DRBG (...) uses a seed that is m bits in length to initiate the generation of *blocksize*-bit pseudorandom strings by performing scalar multiplications on two points in an elliptic curve group, where the curve is defined over a field approximately 2^m in size. *blocksize* has been chosen to ensure full entropy in the output strings; it is a multiple of 8 that is close to but no larger than $m - 16$ (see Annex C.3.2 for details). Selecting an m as small as possible -- subject to the security strength required -- may result in improved performance. For all the NIST curves given in this Standard, $m \geq 163$. Figure 18 depicts the **Dual_EC_DRBG (...)**.

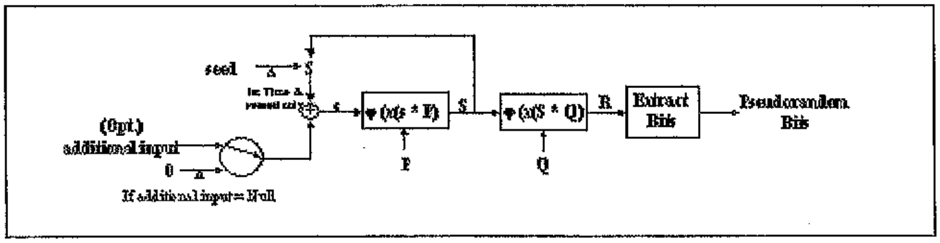

Figure 18: Dual_EC_DRBG (...)

The instantiation of this DRBG requires the selection of an appropriate elliptic curve and curve points specified in Annex A.1 for the desired security strength. The *seed* used to determine the initial value (S) of the DRBG **shall** have entropy that is at least the maximum of 128 and the desired security strength (i.e., *entropy* ≥ **max** (128, *strength*)). The *seed* length **shall** be m bits in length. Further requirements for the *seed* are provided in Section 8.5.

Backtracking resistance is inherent in the algorithm, even if the internal state is compromised. Prediction resistance is also inherent when observed from outside the DRBG boundary. If an application is concerned about the compromise of the hidden state in an instantiation of the **Dual_EC_DRBG(...)**, the state may be infused with new entropy in a number of ways, as discussed in Section 10.3.2.2.5.

When optional additional input (*additional_input)* is used, the value of *additional_input* is arbitrary, in conformance with Section 8.7, but it will be hashed to an *m*-bit string.

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

Table 3 provides guidance for the selection of appropriate curves and hash functions for each desired security strength, together with the associated entropy, seed length and blocksize requirements. Complete specifications for each curve are provided in Annex A.1.

Table 3: Appropriate Dual_EC_DRBG (...) Selections for Desired Security Strengths

Maximum Security Strengths	**Curve**	**Minimum Entropy Requirement**	**Seed length = *m***	**Entropy Input Length (*m*')**	**Block Size**	**Appropriate Hash Functions**
80	B-163	128	163	168	144	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512
80	K-163	128	163	168	144	
80	P-192	128	192	192	176	
112	P-224	128	224	224	208	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512
112	B-233	128	233	240	216	
112	K-233	128	233	240	216	
128	P-256	128	256	256	240	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512
128	B-283	128	283	288	264	
128	K-283	128	283	288	264	
192	P-384	192	384	384	368	SHA-224, SHA-256, SHA-384, SHA-512
192	B-409	192	409	416	392	
192	K-409	192	409	416	392	
256	P-521	256	521	528	504	SHA-256, SHA-384, SHA-512
256	B-571	256	571	576	552	
256	K-571	256	571	576	552	

Comment [ebb1]: Page: 2
Please check this column. We are assuming that the maximum strength that can be supported by a hash function is = to the output block size, if sufficient entropy is obtained.

10.3.2.2 Interaction with Dual_EC_DRBG (...)

10.3.2.2.1 Instantiating Dual_EC_DRBG (...)

Comment [ebb2]: Page: 1
The RNG editing group needs to discuss the philosophy of allowing the user application to determine teh reseeding interval. The hash-based DRBGs don't currently do this. We need to be consistent if it makes sense for a given DRBG.

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

> (*status, state_pointer*) = **Instantiate_Dual_EC_DRBG** (*requested_strength, prediction_resistance_flag , personalization_string , requested_curve_type , reseed_interval, mode*)

as described in Sections 9.5.1 and 10.3.2.3.3, with the addition of the *requested_curve_type* and *reseed_interval* parameters. *requested_curve_type* is used to specify a class of elliptic curves from which the instantiated elliptic curve is to be selected. *reseed_interval* indicates the maximum number of steps that may be taken along the curve before the DRBG must be reseeded.

10.3.2.2.2 Reseeding a Dual_EC_DRBG (...) Instantiation

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

> *status* = **Reseed_ Dual_EC_DRBG_Instantiation** (*state_pointer, additional_input, mode*)

Comment [ebb3]: Page: 3
This was added to allow another avenue of providing entropy or customization.

as described in Sections 9.6.2 and 10.3.2.3.4.

10.3.2.2.3 Generating Pseudorandom Bits Using Dual_EC_DRBG (...)

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

> (*status, pseudorandom_bits*) = **Dual_EC_DRBG** (*state_pointer, requested_strength, requested_no_of_bits, additional_input_string, prediction_resistance_request, mode*)

as described in Sections 9.7.2 and 10.3.2.3.5. The *requested_strength* parameter in the call to **Dual_EC_DRBG (...)** is a failsafe mechanism. The implementation will check that the value requested is not more than that provided by the instantiation, as determined by the call to **Instantiate_Dual_EC_DRBG (...).** A call for greater strength will result in an error condition .

10.3.2.2.4 Removing a Dual_EC_DRBG (...) Instantiation

An application may remove a DRBG instantiation (i.e., release the state space for that instantiation) using the following call:

> *status* = **Uninstantiate_Dual_EC_DRBG** (*state_pointer*)

as described in Sections 9.8 and 10.3.2.3.6.

10.3.2.2.5 Self Testing of the Dual_EC_DRBG (...)

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

status = **Test_Dual_EC_DRBG ( )**

as described in Sections 9.9 and 103.2.3.7.

10.3.2.2.4 Inserting Additional Entropy into the State Using Dual_EC_DRBG (...)

Comment [ebb4]: Page: 4
A generalized discussion of this (i.e., not specific to a given DRBG) has been added as Section 9.10. How much of this do we need (other than, perhaps) some of the details in item 2 ? Does this need a section of its own ?

Additional entropy **may** be inserted into the state of the **Dual_EC_DRBG (...)** in 4 ways:

1. By calling the **Reseed_Dual_EC_DRBG_Instantiation(...)** function at any time. This function always calls the implementation-dependent function **Get_Entropy (...)** for *min_entropy* = **max** (128,*strength*) new bits of entropy, which are added to the state. Section 9.5.2 discusses the **Get_entropy (...)** function.

2. By utilizing the automatic reseeding feature of the **Dual_EC_DRBG(...)**. If *reseed_interval* is set to any positive integer *k* at instantiation, **Reseed_Dual_EC_DRBG_Instantiation(...)** is called automatically whenever *k blocksize* bits of random have been generated since the previous reseeding. As explained above, the reseed function introduces *min_entropy* bits of entropy each time it is invoked. Note that automatic reseeding with *k* = 10,000 is done by default if a *reseed_interval=0* is supplied (see Annex C.3.2). Automatic reseeding can be turned **off** by setting *k* < 0.

3. By setting *prediction_resistance_flag* ≠ 0 at instantiation. If set, any call to **Dual_EC_DRBG(...)** may include a *prediction_resistance_request*, which in turn invokes a call to **Reseed_Dual_EC_DRBG_Instantiation()** before new random is produced. (Comment: Frequent calls to the **Get_Entropy()** function may cause **severe** performance degradation with this or any DRBG.)

4. By supplying an *additional_input_string* on any call to **Dual_EC_DRBG(...)** for random bits.

10.3.2.3 Specifications

10.3.2.3.1 General

The instantiation of **Dual_EC_DRBG (...)** consists of selecting an appropriate elliptic curve and point pairing from Annex A.1 and obtaining a *seed* that is used to determine an initial value (*S*). The state consists of:

Comment [ebb5]: Page: 4
Note that the usage_class was removed.

1. A counter (*reseed_counter*) that indicates the number of blocks of random produced by the **Dual_EC_DRBG (...)** during the current instance and since the previous reseeding.

2. A *reseed_interval* specifies the frequency, in blocks of *blocksize* bits of random produced, at which automatic reseeding of the **Dual_EC_DRBG (...)** occurs.

3. A value (S) that determines the current position on the curve E,
4. The elliptic curve domain parameters (*curve_type*, *m*, *p*, *a*, *b*, *n*), where *curve_type* indicates a prime field F_p, or a pseudorandom or Koblitz curve over the binary field F_2^m ; *a* and *b* are two field elements that define the equation of the curve, and *n* is the order of the point *G*,
5. Two points *P* and *Q* on the curve; the generating point *G* specified in FIPS 186-2 for the chosen curve will be used as *P*,
6. The security *strength* provided by the instance of the DRBG,
7. A *prediction_resistance_flag* that indicates whether prediction resistance is required by the DRBG., and
8. A record of the seeding material in the form of a one-way function that is performed on the *entropy_input* for later comparison with new *entropy_input* when the DRBG is reseeded.

10.3.2.3.2 Dual_EC (...) Variables

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

a, *b* Two field elements that define the equation of the curve.

additional_input_string Optional additional input. A byte array that may be provided on any call for random bits or during reseeding. The string will be hashed to *m* bits using **Hash_df (...)**. See Section 9.5.4.2

additional_input The hashed bitstring derived from the optional *additional_input_string.*

blocksize The number of bits output by a single step of the **Dual_EC_DRBG(...)**. The precise value depends on the curve chosen, but is always a multiple of 8 near *m*-16. (See Annex C.3.2 and Section 10.3.2.3.3)

curve_type Either 0 (*Prime_field_curve*),1 (*Random_binary_curve*),or 2 (*Koblitz_curve*), indicating a curve over a prime field, a random binary curve, or a Koblitz curve, respectively. The default curve type is 0 (i.e., mod *p* will be used).

E An elliptic curve defined over F_p or F_2^m.

entropy_input The bits containing entropy that are used to determine *seed_material* and generate a *seed.*

f The cofactor of the curve: 1 for all prime field curves, 2 or 4 for the binary curves. Comment: This value will be implicit from the *curve_type* and *a*.

Find_state_space (*mode*)

A function that finds an unused *state* in the state space. See Section 9.5.3.

G A generating point of prime order *n* on the curve *E*.

Get_entropy (*min_entropy,min_length, max_length, mode*)

A function that acquires a string of bits from an entropy input source. The parameters indicate the minimum entropy to be provided in the returned bit string, and the limits between which the length of that string must lie (i.e., *min_length* and *max_length*). **Dual_EC_DRBG (....)** will always specify *min_length* = *max_length* = *m*. *mode* indicates whether the function is called during normal operation or during testing. Also, see Section 9.5.2.

Comment [ebb6]: Page: 1
Do they really need to be the same ? See the comment in Section 10.3.2.3.3, step 13.

Hash (*hash_input*) An Approved hash function that returns a bitstring whose input *hash_input* may be any multiple of 8 bits in length.

Hash_df (*hash_input, output_len*)

A function to distribute the entropy in *hash_input* to a bitstring *output_len* long. The function **Hash (...)** is used to do this. *hash_input* may be any multiple of 8 bits in length; *output_len* is arbitrary. See Section 9.5.4.2.

i A temporary value that is used as a loop counter.

Invalid_state_pointer An illegal value for the *state_pointer*.

len (*A*) The length in bits of the string *A*.

m Length in bits of the internal state *S*; the curve is defined over a field with approximately 2^m elements.

max (*A*, *B*) The maximum of the values *A* and *B*.

max_length The maximum length of the *entropy_input*.

max_no_of_states The maximum number of states and instantiations that an implementation can handle.

min_entropy A value used in the request to **Get_entropy (...)** to indicate the minimum entropy to be provided.

Comment: In fact, the value of *strength* is used in this determination, and *strength* is always at least *requested_strength.*

min_length The minimum length of the *entropy_input*.

mode An indication of whether a request for entropy input is for normal operation or for testing. For normal operation, *mode* = 0 = *Normal_operation*. See Section 9.9.2.1 for testing values.

n The order of the generating point *G* on the curve.

Null The null (empty) string.

old_transformed_entropy_input

A record of the *entropy_input* used in the previous instance of the DRBG.

p The modulus when *curve_type* = 0 (*Prime_field_curve*); an *m*-bit prime.

P, *Q* Two points on the elliptic curve *E*, such that each generates a large cyclic subgroup on *E*. The generating point *G* will be used as *P*.

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

personalization_string A byte array that can provide additional assurance of seed uniqueness at instantiation.

prediction_resistance_flag

An instantiation flag indicating whether or not prediction resistance is to be provided by the DRBG. If set to 1 (*Allow_prediction_resistance*), prediction resistance requests may be made during calls for random bits. If set to 0 (*No_prediction_resistance*), later requests for prediction resistance will return an error message.

prediction_resistance_request

Setting *prediction_resistance_request* = 1 (*Provide_prediction_resistance*) at a call to **Dual_EC_DRBG(...)** specifies that **Reseed_Dual_EC_DRBG_Instantiation(...)** is to be called before new random is produced. If *prediction_resistance_flag* is not set to *Allow_prediction_resistance* during the call to **Instantiate_Dual_EC_DRBG(...)**, the request will return an error message.

pseudorandom_bits The pseudorandom bits produced by the DRBG.

R
A value from which pseudorandom bits are extracted.

requested_curve_type
The *curve_type* can be specified as input to **Instantiate_Dual_EC_DRBG (...)**; if none is requested, the default value of 0 (*Prime_field_curve*) is assigned.

requested_no_of_bits
The number of pseudorandom bits to be returned on a call to **Dual_EC_DRBG (...)**.

requested_strength
The security *strength* of the bits requested from the DRBG. The bits returned may have more than *requested_strength* bits of security, but never less.

reseed_counter
A count of the number of iterations of the of **Dual_EC_DRBG (...)** since the last reseeding.

reseed_interval
The maximum number of steps taken along the curve before the DRBG must be reseeded. The default value 10,000 is recommended (see Annex C.3.2) and may be selected by setting the *reseed_interval* input parameter to the instantiation process to 0 (*Use_default_reseed_interval*). If *reseed_interval* < 0, automatic reseeding will not be performed.

s
A temporary value.

S
A value that is initially determined by a *seed*, but assumes new values during each request of pseudorandom bits from the DRBG.

seed_material
The seed used to derive the initial value of *S*.

state(*state_pointer*)
An array of states for different DRBG instantiations. A *state* is carried between DRBG calls. For the **Dual_EC_DRBG** (...), the *state* for an instantiation is defined as *state* (*state_pointer*) = {*reseed_counter*, *reseed_interval*, *S*, *curve_type*, *p*, *a*, *b*, *n*, *P*, *Q*, *strength*, *prediction_resistance_flag* , *transformed_entropy_input*}. A particular element of the *state* is specified as *state*(*state_pointer*).*element*, e.g., *state* (*state_pointer*).*S*. Comment : *p* is only needed by the *curve_type*=0 curves (*Prime_field_curve*).

status
The *status* returned from a function call, where *status* = "Success" or a failure message.

strength
The maximum strength of an instance of the DRBG (i.e., 80, 112, 128, 192 or 256).

temp
A temporary value.

temp_input A temporary value.

transformed_entropy_input

A one-way transformation of the *entropy_input* for the **Hash_DRBG (...)** instance.

Truncate (*bits, in_len, out_len*)

A function that inputs a bit string of *in_len* bits, returning a string consisting of the leftmost *out_len* bits of input. If *in_len* < *out_len*, the input string is padded on the right with (*out_len* - *in_len*) zeroes, and the result is returned.

x(*A*) The *x*-coordinate of the point *A* on the curve *E*.

φ A mapping from field elements to non-negative integers that takes the bit vector representation of a field element and interprets it as the binary expansion of an integer. Section 10.3.2.3.5 has the details of this mapping.

* Scalar multiplication of a point on the curve.

10.3.2.3.3 Instantiation of Dual_EC_DRBG (...)

The following process or its equivalent **shall** be used to instantiate the **Dual_EC_DRBG (...)** process. Let **Hash (...)** be an Approved hash function for the security strengths to be supported. If the DRBG will be used for multiple security strengths, and only a single hash function will be available, that hash function **shall** be suitable for all supported security strengths (see Table 3 and SP 800-57).

Instantiate_Dual_EC_DRBG (...):

Input: integer (*requested_strength, prediction_resistance_flag ,personalization_string, requested_curve_type , reseed_interval, mode*)

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

Comment [ebb7]: Page: 9
The usage_class is no longer an input ; a state_pointer is output.

Process:

1. If (*requested_strength* > the maximum security strength that can be provided by the implementation (see Table 3)), then **Return** ("Invalid *requested_strength*", *Invalid_state_pointer*).

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

   Comment : Find an empty state in the state space for the instantiation.

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

```
4. If (status ≠ "Success"), the Return (status, Invalid_state_pointer).

                                        Comment : Determine an mthat is appropriate
                                        for the requested_strength; this will depend
                                        on curve_type.

5. If (requested_curve_type = Prime_field_curve), then

                                        Comment : choose one of the prime field
                                        curves. There is no NIST curve with $m = 160$.
                                        The smallest mod p curve in FIPS 186-2 is for
                                        $m = 192$. Therefore, when the DRBG is
                                        instantiated with a nominal strength of 80, the
                                        actual strength is 96.

    If (requested_strength $\le 80$), then {strength = 80,  $m = 192$}
    Else if (requested_strength $\le 112$), then {strength = 112,  $m = 224$}
        Else if (requested_strength $\le 128$), then {strength = 128,  $m = 256$}
            Else if (requested_strength $\le$ 192), then {strength = 192 , $m = 384$}
                Else if (requested_strength $\le 256$), then {strength = 256,  $m$ =
                    521}

                                        Comment: There is no NIST curve with $m$ =
                                        512 .

6. If (requested_curve_type ≠ Prime_field_curve), then

                                        Comment: choose one of the binary or
                                        Koblitz curves.

    If (requested_strength $\le 80$), then {strength = 80,  $m = 163$}
    Else if (requested_strength $\le 112$), then {strength = 112,  $m = 233$}
        Else if (requested_strength $\le 128$), then {strength = 128,  $m = 283$}
            Else if (requested_strength $\le$ 192), then {strength = 192 , $m = 409$}
                Else if (requested_strength $\le 256$), then {strength = 256,  $m$ =
                    571.

                                        Comment: Select the appropriate curve. For
                                        the binary and Koblitz curves, $p = 0$.

7. If (curve_type = Prime_field_curve), then select elliptic curve P-m
   Else if (curve_type = Random_binary_curve), then select elliptic curve B-m and
       set $p = 0$
```

Else if (*curve_type* = *Koblitz_curve*), then select elliptic curve K-*m* and set *p* = 0.

8 Set the point *P* to the generator *G* for the curve, and set *n* to the order of *G*.

9. Set the corresponding point *Q* from Annex A.1.

10. Set the *blocksize*—the number of bits to use for each iteration of the **Dual_EC_DRBG(...)**. As explained in Annex C.3.2, this number depends on the *curve_type* and its size *m*. Only the rightmost *blocksize* bits of each block produced are output; the others are discarded. The formula for *blocksize* is [smallest multiple of 8 larger than m-(13+$\log_2(f)$)]. The following table summarizes the *blocksize* calculation, in the format [NIST curve : *blocksize*]:

Prime_field_curve	*Random_binary_curve*	*Koblitz_curve*
P-192 : 176	B-163 : 144	K-163 : 144
P-224 : 208	B-233 : 216	K-233 : 216
P-256 : 240	B-283 : 264	K-283 : 264
P-384 : 368	B-409 : 392	K-409 : 392
P-521 : 504	B-571 : 552	K-571 : 552

Comment: Set the automatic reseeding interval. Automatic reseeding will **not** occur if a negative value for *reseed_interval* is provided.

11. If (*reseed_interval* = 0), then *reseed_interval* = 10,000.

Comment: Request *entropy_input* with the desired entropy and bitlength (the smallest multiple of 8 at least as large as *m*).

12. *min_entropy* = **max** (128, *strength*).

13. *min_length* = *max_length* = $8 \times \lceil m/8 \rceil$.

Comment [ebb8]: Page: 11
Couldn't the *max_length* be larger ; the *entropy_input* source may not have full entropy. Note that step 17 hashes down to the desired size for *S*.

14. (*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_length*, *max_length*, *mode*).

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

16. *seed_material* = *entropy_input* || *personalization_string*.

Comment: Use a hash function to ensure that the entropy is distributed throughout the bits, and *S* is *m* bits in length.

17. S = **Hash_df** (*seed_material*, *m*).

Comment: Perform a one-way function on the *entropy_input* for later comparison.

18. *transformed_entropy_input* = **Hash** (*entropy_input*).

Comment [ebb9]: Page: 12
We need to save the result from the Get_entropy function for later comparison with the next result to determine if the entropy input source has failed. We don't want to hash the seed material, since it includes the personalization string. Note that I made the same error.

19. *reseed_counter* = 0.

Comment: *reseed_counter* is incremented every *blocksize* bits.

Comment: Save all state information.

20. *state*(*state_pointer*) = {*reseed_counter*, *reseed_interval*, *S*, *curve_type*, *m*, *p*, *a*, *b*, *n*, *P*, *Q*, *strength*, *prediction_resistance_flag* , *transformed_entropy_input*}.

21. **Return** ("Success", *state_pointer*).

10.3.2.3.4 Reseeding of a Dual_EC_DRBG (...) Instantiation

The following process or its equivalent **shall** be used to reseed the **Dual_EC_DRBG (...)** process, **after** it has been instantiated.

Reseed_Dual_EC_DRBG_Instantiation (...):

Input: integer *state_pointer*, string *additional_input_string*, integer *mode*.

Output: string *status*.

Process:

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

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

2. *S* = *state*(*state_pointer*).*S*, *m* = *state*(*state_pointer*).*m*, *strength* = *state*(*state_pointer*).*strength*, *old_transformed_entropy_input* = *state*(*state_pointer*).*transformed_entropy_input*.

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

3. *min_entropy* = **max** (128, *strength*).

4. *min_length* = *max_length* = $8 \times \lceil m/8 \rceil$

5. (*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_ length*, *max_length*, *mode*).

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

```
                         Comment: Perform a one-way function on the
                         entropy_input for comparison.

 7. transformed_entropy_input = Hash (entropy_input).

                         Comment: Check for a viable entropy source.

 8. If (transformed_entropy_input = old_transformed_entropy_input), then

      If (mode = Normal_operation), then Abort_to_error_state
         ("Entropy_input source failure")

      Else Return ("Entropy_input source failure").

                         Comment: Combine new entropy_input with
                         the old state and any additional_input.

 9. seed_material = pad8 (S) || entropy_input || additional_input_string.

10. S = Hash_df (seed_material, m).

                         Comment: Update the changed values in the
                         state.

11. state.S = S.

12. state.transformed_entropy_input = transformed_entropy_input.

13. state.reseed_counter = 0.

14. Return ("Success").
```

Comment [ebb10]: Page: 13
We need to save the result from the Get_entropy function for later comparison with the next result to determine if the entropy input source has failed. We don't want to hash the seed material, since it includes the personalization string. Note that I made the same error.

10.3.2.3.5 Generating Pseudorandom Bits Using Dual_EC_DRBG (...)

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

Dual_EC_DRBG (...):

Comment [ebb11]: Page: 13
If there is no reseeding capability, do we need to insert a check to see if the DRBG has maxed out and then return an error ? This could be done, for example, after step 2.

Input: integer (*state_pointer, requested_strength, requested_no_of_bits, additional_input_string, prediction_resistance_request, mode*).

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

Process:

```
1. If ((state_pointer > max_no_of_states) or (state (state_pointer) = {0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0 0, -1, Null}), then Return ("State not available for the
   state_pointer", Null).

                         Comment: Get the appropriate state
                         values for the indicated state_pointer.

2. S = state(state_pointer).S, m = state(state_pointer).m, strength =
   state(state_pointer).strength, P = state(state_pointer).P, Q =
   state(state_pointer).Q, reseed_counter = state(state_pointer).reseed_counter,
    reseed_interval = state(state_pointer).reseed_interval;
    prediction_resistance_flag = state(state_pointer).prediction_resistance_flag.

                                        Comment: Check that the requested_strength
                                        is not more than that provided by this
                                        instantiation.

 3. If (requested_strength > strength), then Return ("Invalid requested_strength",
    Null).

 4. If ((prediction_resistance_request = Provide_prediction_resistance) and
    (prediction_resistance_flag = No_prediction_resistance)), then Return
    ("Prediction resistance capability not instantiated", Null).

                                        Comment: Check for supplied additional
                                        input. This will be added to the state on the
                                        first iteration only.

 5. If (additional_input_string = Null) then additional_input = 0

                                        Comment: additional_input set to m zeroes.

    Else additional_input = Hash_df (pad8 (additional_input_string), m).

                                        Comment: Hash to m bits.

                                        Comment: If a prediction resistance request
                                        has been made, instill new entropy with a call
                                        to reseed the Dual_EC_DRBG(...).
                                        Reseed_Dual_EC_DRBG (...) resets
                                        reseed_counter to 0

 6. If (prediction_resistance_request = Provide_prediction_resistance), then

    6.1 status = Reseed_Dual_EC_DRBG_Instantiation (state_pointer, Null,
        mode).

    6.2 If (status $\neq$ "Success"), then Return (status, Null).

                                        Comment: Produce requested_no_of_bits,
                                        blocksize at a time:

 7. temp = the Null string.

 8. i = 0.

                                        Comment: Determine if reseeding is required.
                                        The reseeding process resets reseed_counter
                                        to 0.

 9. If ((reseed_interval > 0) and (reseed_counter = reseed_interval)), then
```

Comment [ebb12]: Page: 14
Alternatively, this could be essentially be replaced by a call to the get_entropy routine and checking that the entropy_input isn't the same as the last time. Your call.

```
        9.1 status = Reseed_Dual_EC_DRBG_Instantiation (state_pointer, Null,
            mode).

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

    10. $s = S \oplus additional\_input$.

    11. $S = \varphi(x(s * P))$.        Comment: s is to be interpreted as an m- bit
                                        unsigned integer. To be precise, when
                                        curve_type = Prime_field_curve, s should be
                                        reduced mod n; the operation * will effect
                                        this. S is an m-bit number. See footnote [1].

    12. $R = \varphi(x(S * Q))$.        Comment: R is an m-bit number. See footnote
                                        [1].

    13. temp = temp || (rightmost blocksize bits of R ).

    14. additional_input=0              Comment: m zeroes; additional_input_string
                                        is added only on the first iteration.

    15. reseed_counter = reseed_counter + 1.

    16. $i = i + 1$.

    17. If (len (temp) < requested_no_of_bits), then go to step 9.

    18. pseudorandom_bits = Truncate (temp, $i \times blocksize$, requested_no_of_bits).

                                             Comment: Update the changed values
                                             in the state.

    19. state.S = S.

    20. state.reseed_counter = reseed_counter.
```

[1] The precise definition of $\varphi(x)$ used in steps 11 and 8 depends on the field representation of the curve points. In keeping with the convention of FIPS 186-2, the following elements will be associated with each other:

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

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

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

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

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

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

```
    21. Return ("Success", pseudorandom_bits).
```

10.3.2.3.6 Removing a Dual_EC_DRBG (...) Instantiation

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

Uninstantiate_Dual_EC_DRBG (...):

```
Input: integer state_pointer.

Output: string status.

Process:

    1. If (state_pointer > max_no_of_states), then Return ("Invalid state_pointer").

    2. state(state_pointer) = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 0, -1, Null}.

    3. Return ("Success").
```

10.3.2.3.6 Self Testing of the Dual_EC_DRBG (...)

Self testing **shall** be performed on the **Dual_EC_DRBG (...)** processes contained within a DRBG boundary using the specifications in Section 9.9.

[Differences to be determined].

10.3.2.3.7 Implementation Considerations

In deference to the defacto standard of using character arrays as inputs and outputs for hash implementations, the **Dual_EC_DRBG (...)** pads bitstrings to byte boundaries, and requests its seed material to comply as well. This is the reason for the variable *min_length* and the function **pad8** (...).

Comment [ebb13]: Page: 16
Need to correct this.

Comment [ebb14]: Page: 16
This subject was handled differently for the hash_based DRBGs. We need to decide the best way to do this.

10.3.2.4 Generator Strength and Attributes

The particular curve to be used **shall** be based on *strength*, which is selected from one of five security levels and **shall** always be at least *requested_strength*. The curves and associated security levels are those given in FIPS 186-3; they are meant to correspond to the strengths of various standard symmetric encryption algorithms.

For each security strength, there are three curves associated with each security level, one defined over a prime field GF(p) and two over a binary field GF(2^m), where $2^m \approx p$. The mod p curves, assigned *curve_type* 0 (*Prime_field_curve*), are used by default. Any of the three curves may be used for the security level.

Initial seeding is accomplished with a call to **Get_entropy(...)**, which returns a bitstring of a specified length and entropy. The **Dual_EC_DRBG (...)** specifies **max** (128, *strength*) bits of entropy.

10.3.2.4 Reseeding and Rekeying

The reseeding process is specified in Section 10.3.2.3.4 . Automatic reseeding is done by default after each 10,000 *blocksize* bits of random are generated. The frequency may be changed by providing a positive value for *reseed_interval*, or the feature may be disabled by setting *reseed_interval* < 0 at instantiation. Alternatively, or in addition, a call to **Reseed_Dual_EC_DRBG_Instantiation(...)** can be made at any time.

The **Dual_EC_DRBG** (...) is not keyed per se; however, the *additional_input* and *personalization_string* features may be used to effect keying, if desired.

Comment [ebb15]: Page: 17
Do we want to make this a general statement for all the DRBGs ? Or not say it at all ?

Comment [ebb16]: Page: 17
Need to decide if this belongs here. Also, do we really need these sections ?

10.3.3 Micali-Schnorr Deterministic RBG (MS_DRBG)

10.3.3.1 Discussion

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

Comment [ebb17]: Page: 18
Can this be removed ?

The Micali-Schnorr generator **MS_DRBG(...)** uses the same e and n to produce many more random bits per iteration, while removing the incestuous relationship between the state sequence and the output bits. Each $y_i \in [0,n)$ is viewed as the concatenation $s_i \parallel z_i$ of an r-bit number s_i and a k = lg(n)-r bit number z_i. The s_i are used to propagate the integer sequence $y_{i+1} = s_i^e \ mod\ n$; the z_i are output as random bits. r must be at least 2*min{*strength*, lg(n)/e}, where *strength* is the desired security strength of the generator, and $e \geq 3$. (See Section 10.3.3.3.2.) A random r-bit *seed* s_0 is used to initialize the process. Figure 19 depicts the **MS_DRBG(...)**.

Comment [ebb18]: Page: 18
Can this be reworded ?

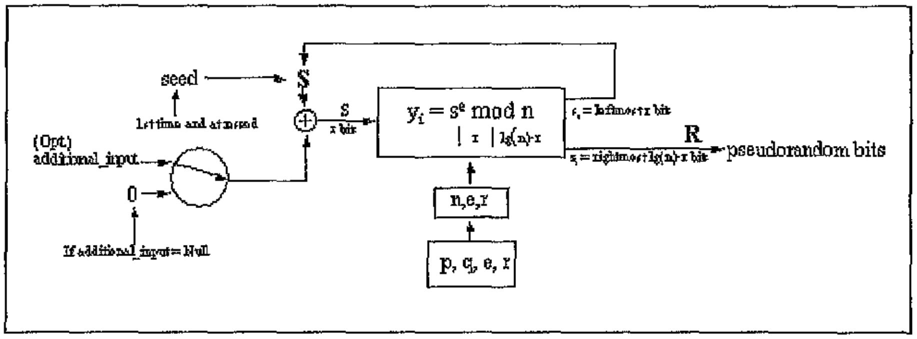

Figure 19: MS_DRBG (...)

The **MS_DRBG(...)** is cryptographically secure under the assumption that sequences of the form $s^e \ mod\ n$ are statistically the same as sequences of integers in Z_n. This assumption is stronger than requiring the intractability of the RSA problem.

For **MS_DRBG (...)**, the s values are assumed to be r-bit integers, and "statistically the same" means indistinguishable by any polynomial-time algorithm. Accepting the stronger assumption allows k to be a significant percentage of lg(n).

The lengths r and k , the RSA modulus n, and the value of the exponent e are variable within the bounds described below. The bounds are based on the desired *strength* of bits produced. For maximum efficiency, e **should** be kept small and k **should** be large. The k

bits generated at each step are concatenated to form pseudorandom bit strings of any desired length.

Seeding material is provided by the implementation-dependent function **Get_entropy(…)**. The mininum entropy required from this function will be set to **max** (128, *strength*), per Section 9.3.

Backtracking resistance is inherent in the RSA algorithm, even if the internal state is compromised. Prediction resistance is inherent when observed from outside the DRBG boundary. If an application is concerned about the compromise of the hidden state in an instantiation of the **MS_DRBG(…)**, the state may be infused with new entropy in a number of ways, as discussed in Sections 8.5 and 9.3.

When optional additional input (*additional_input)* is used, the value of the *additional_input* is arbitrary, in conformance with Section 8.7. It will be hashed to an *r*-bit string.

10.3.3.2 Interaction with MS_DRBG (...)

10.3.3.2.1 Instantiating MS_DRBG (...)

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

> (*status*, *state_pointer*) = **Instantiate_MS_DRBG** (*requested_strength,*
> *prediction_resistance_flag*, *personalization_string, use_random_primes* ,
> *requested_e* , *requested_k* , *reseed_interval*, *mode*)

as described in Section 9.5.1, with the addition of the *use_random_primes, requested_e* , *requested_k* , and *reseed_interval* parameters. The application may request a specific RSA exponent *e*, or specify the output size *k* of bits produced on an iteration of **MS_DRBG (…)**. *reseed_interval* indicates the maximum number of *k*-bit output blocks that may be produced before the DRBG must be reseeded. Alternatively, default values for *e*, *k*, and *reseed_interval* will be used if zero values are supplied.

Setting *use_random_primes* = 1 instructs the implementation to generate random primes using an Approved method at instantiation; otherwise, default values appropriate for the *requested_strength* will be selected from Annex A.2.

10.3.3.2.2 Reseeding a MS_DRBG (...) Instantiation

When a DRBG instantiation requires explicit reseeding (see Section 9.6), the DRBG **shall** be reseeded using the following call:

> *status* = **Reseed_MS_DRBG_Instantiation**(*state_pointer*, *additional_input_string*,
> *mode*).

10.3.3.2.3 Generating Pseudorandom Bits Using MS_DRBG (...)

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

(*status*, *pseudorandom_bits*) = **MS_DRBG** (*state_pointer*, *requested_strength*, *requested_no_of_bits*, *additional_input_string*, *prediction_resistance_request*, *mode*)

as described in Section 9.7.2. In particular, a request for higher strength than was set at instantiation will result in an error.

10.3.3.2.4 Removing an MS_DRBG (...) Instantiation

An application may remove a DRBG instantiation (i.e., release the state space for that instantiation) using the following call:

status = **Uninstantiate_MS_DRBG** (*state_pointer*)

as described in Section 9.8.

10.3.2.2.5 Self Testing of the MS_DRBG (...)

An **MS_DRBG**(...) implementation is tested at power up and on demand using the following call:

status = **Test_MS_DRBG** (.)

as described in Section 9.9.

10.3.3.2.6 Inserting Additional Entropy into the State of MS_DRBG (...)

Comment [ebb19]: Page: 1
See the comment for Section 10.3.2.2.4.

Additional entropy **may** be inserted into the state of the **MS_DRBG (...)** in 4 ways:

1. By calling the **Reseed_MS_DRBG_Instantiation(...)** function at any time. This function always calls the implementation-dependent function **Get_entropy (...)** for *min_entropy* = max(128,*strength*) new bits of entropy, which are added to the state. Section 9.5.2 discusses the **Get_entropy (...)** function.

2. By utilizing the automatic reseeding feature of the **MS_DRBG(...)**. The *reseed_interval* may be set to any positive value *j* at instantiation. **Reseed_MS_DRBG_Instantiation (...)** is called automatically whenver *j*k* bits of random have been output since the previous reseeding. As explained above, the **Reseed_MS_DRBG_Instantiation (...)** function introduces *min_entropy* bits of entropy each time it is invoked. If a 0 value is provided as the *reseed_interval* during instantiation, *reseed_interval* defaults to 50,000. Automatic reseeding is turned **off** by setting *j* < 0.

3. By setting *prediction_resistance_flag* = 1 (*Allow_prediction_resistance*) at instantiation. If set, any call to **MS_DRBG(...)** may include a *prediction_resistance_request*, which in turn invokes a call to **Reseed_MS_DRBG_Instantiation()** before new random is produced. Note that frequent calls to the **Get_entropy (...)** function may cause significant performance degradation with this or any DRBG.

4. By supplying an *additional_input_string* on any call to **MS_DRBG()** for random bits.

10.3.3.3 Specifications

10.3.3.3.1 General

During the instantiation of **MS_DRBG (...)**, the M-S parameters *n*, *e*, *r*, and *k* are selected as described in Section 10.3.3.3.3, and a random initial seed s_0 is obtained. Each of these become part of the internal *state* of the DRBG. The state consists of:

1. The M-S parameters *n, e, r* and *k*.
2. A seed $s \in [0,2^r)$ that is updated during each request for pseudorandom bits.
3. The security *strength* provided by the instance of the DRBG.
4. The mininum entropy needed from a call to **Get_entropy(...)** for seeding material. The value of *min_entropy* will be set to **max** (128, *strength*), per Section 9.3.
5. A *reseed_interval* may be provided that will automatically reseed the **MS_DRBG (...)** whenever *reseed_interval* iterations (*k*-bit blocks) have been made since the previous reseeding.
6. A counter (*reseed_counter*) that indicates the number of blocks of random produced by **MS_DRBG (...)** during the current instance since the previous reseeding.
7. A *prediction_resistance_flag* that indicates whether prediction resistance is required by the DRBG., and
8. A record of the seeding material in the form of a one-way function that is performed on the *entropy_input* for later comparison with new *entropy_input* when the DRBG is reseeded.

Comment [ebb20]: Page: 1
This should probably refer to *S*, which is used in the pseudocode. I'm not sure how you want to word this.

10.3.3.3.2 MS_DRBG (...) Variables

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

additional_input_string Optional additional input. A byte array that may be provided on any call for random bits. The string will be hashed to *r* bits using **Hash_df (...)**.

additional_input The hashed bitstring derived from the optional *additional_input_string*.

e A positive integer that is used as an RSA exponent.

entropy_input The bits containing entropy that are used to determine *seed_material* and generate a *seed*.

Find_state_space (*mode*) A function that finds an unused state in the state space. See Section 9.5.3.

gcd (*x,y*) The greatest common divisor of the integers *x* and *y*.

Get_entropy (*min_entropy, min_length, max_length, mode*)

A function that acquires a string of bits from an entropy input source. The parameters indicate the minimum entropy to be provided in the returned bit string, and the limits between which the length of that string must lie. The **MS_DRBG (...)** will always specify *min_length* = *max_length* = *r*. *mode* indicates whether the function is called during normal operation or during testing.

Get_random_modulus (lg(*n*), *e*, *n*)

Comment [ebb21]: Page: 1
Need a definition/specification for this.

Invalid_state_pointer An illegal value for the *state_pointer*.

k The number of bits generated at each iteration of **MS_DRBG (...)**; as an implementation convenience, this will always be a multiple of 8 bits.

Hash (*hash_input***)** An Approved hash function that returns a bitstring whose input *hash_input* may be any multiple of 8 bits in length.

Hash_df (*hash_input, output_len***)**

A function to distribute the entropy in *hash_input* to a bitstring *output_len* long. The function **Hash (...)** is used to do this. *hash_input* may be any multiple of 8 bits in length ; *output_len* is arbitrary.

i A temporary value that is used as a loop counter.

lg(*n*) The number of bits in the binary representation of *n ;* it is selected from Table 4 in Section 10.3.3.3.3 based on the requested security strength.

max (*A*, *B*) The maximum of the values *A* and *B*.

max_length The maximum length of the *entropy_input*.

max_no_of_states The maximum number of states and instantiations that an implementation can handle.

min_entropy A value used in the request to **Get_entropy (...)** to indicate the minimum entropy to be provided for seeding material. Comment: In fact, the value of *strength* is used in this determination, and *strength* is always at least *requested_strength.*

min_length The minimum length of the *entropy_input*.

mode An indication of whether a request for *entropy_input* is for normal operation or for testing. For normal operation, *mode* = 0 (*Normal_operation*). See Section 9.9.2.1 for testing values.

M-S parameters *n*, *e*, *r*, *k*

n The RSA modulus; the product of two distinct large primes *p* and *q*.

Null A null (empty) string.

old_transformed_entropy_input

A record of the *entropy_input* obtained during the previous instance of the DRBG.

p, *q* Prime numbers generated using an Approved algorithm, e.g., as defined in ANS X9.31, Annex B. These will be randomly generated at initialization if *use_random_primes* is set to 1. Otherwise, the default modulus of an appropriate size will be used.

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

personalization_string A byte array that can provide additional assurance of seed uniqueness at instantiation.

prediction_resistance_flag

An instantiation flag indicating whether prediction resistance is to be provided by the DRBG. If set to 1 (*Allow_prediction_resistance*), prediction resistance requests may be made during calls for random bits. If set to 0 (*No_prediction_resistance*), later requests will return an error message.

prediction_resistance_request

Comment [ebb22]: Page: 23
This may not make sense.

Setting *prediction_resistance_request* = 1 (*Provide_prediction_resistance*)at a call to **MS_DRBG(…)** specifies that **Reseed_MS_DRBG_Instantiation(…)** is to be called before new random is produced. If *prediction_resistance_flag* is not set to *Allow_prediction_resistance* during the call to **Instantiate_MS_DRBG()**, the request will return an error message.

pseudorandom_bits The pseudorandom bits produced by the DRBG.

r Bit length of the seeds; $r = \lg(n) - k$. Comment: *r* will always be a multiple of 8 bits.

requested_e Requested RSA exponent *e*; a value of 0 indicates that the default value is to be used.

requested_k Requested size *k* of each output string; a value of 0 indicates that the default value is to be used.

reseed_counter An integer count of the number of iterations of the of **MS_DRBG (...)** since the last reseeding.

reseed_interval The maximum number of steps taken before the DRBG must be reseeded. The default value 50,000 is recommended (see Annex C.3.2) and is assigned if *reseed_interval* = 0 is provided when an instantiation is requested. If *reseed_interval* < 0, automatic reseeding will not be performed.

S A value that is initially determined by a seed, but assumes new values during each request of pseudorandom bits from the DRBG.

R A value from which pseudorandom bits are extracted.

s_i The state, or seed, of the generator at the *i*-th iteration; an integer, $s_i \in [0, 2^r)$

s_0 Random initial *r*-bit seed.

Comment [ebb23]: Page: 1
These are not specified in the pseudocode. What should be done with them ?

seed_material The seed used to derive the initial value of *S*.

state(*state_pointer*) An array of states for different DRBG instantiations. A *state* is carried between DRBG calls. For the **MS_DRBG (...)**, the *state* for an instantiation is defined as *state*(*state_pointer*) = {*reseed_counter*, *reseed_interval*, *S*, *n*, *e*, *r*, *k*, *strength*, *min_entropy* , *prediction_resistance_flag*, *transformed_seed*}. A particular element of the *state* is specified as *state.element*, e.g., state(*state_pointer*).*S*

status The *status* returned from a function call, where *status* = "Success" or a failure message.

strength The security *strength* of the bits requested from the DRBG. It will always be at least *requested_strength*. For efficiency, the smallest modulus size lg(*n*) providing *requested_strength* bits of security will be selected from Table 4 in Section 10.3.3.3.3.

transformed_entropy_input

A record of the *entropy_input* used in the current instance of the DRBG.

Truncate (*bits, in_len, out_len*)

A function that inputs a bit string of *in_len* bits, returning a string consisting of the leftmost *out_len* bits of input. If *in_len* < *out_len*, the input string is returned padded on the right with *out_len* – *in_len* zeroes.

use_random_primes If *use_random_primes* = 1 (*Use_random_primes*), random primes of size ½ lg(n) will be generated at initialization, using an Approved algorithm, and having entropy at least *min_entropy*. If *use_random_primes* = 0 (*Random_primes_not_required*), the appropriate modulus from Annex A.2 **shall** be used.

y_i An integer , $y_i \in [0,n)$. $y_i = s_i \,||\, z_i$.

z_i k-bit output of **MS_DRBG (...)** at iteration i.

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

10.3.3.3.3 Selection of the M-S parameters

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

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

1. $1 < e < \phi(n)$; gcd(e, $\phi(n)$) = 1. Comment: ensures that the mapping $s \rightarrow s^e$ mod n is 1-1.
2. $re \geq 2^*\mathrm{lg}(n)$. Comment: ensures that the exponentiation requires a full modular reduction.
3. $r \geq 2$**strength*. Comment: protects against a tableization attack.
4. k,r are multiples of 8. Comment: an implementation convenience.
5. $k \geq 8$; $r + k = \mathrm{lg}(n)$. Comment: all bits are used.
6. $n = p^*q$. Comment: strong [as in X9.31], secret primes .

The M-S parameters are determined in this order:

1. The size of the modulus lg(n) is set first. It **shall** conform to the values given in Table 4 for the requested security *strength*.

Table 4 : Appropriate MS_DRBG (...) Selections

Bits of Security	RSA modulus size	lg(lg(n)) = # of hard bits	Appropriate Hash Functions
80	lg(n) = 1024	10	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512
112	lg(n) = 2048	11	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512
128	lg(n) = 3072	11	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512
192	lg(n) = 7680	12	SHA-224, SHA-256, SHA-384, SHA-512
256	lg(n) = 15360	13	SHA-256, SHA-384, SHA-512

2. The RSA exponent e. The implementation **should** allow the application to request any odd integer e in the range $1 < e < 2^{\lg(n)-1} - 2*2^{½\lg(n)}$. [Comment: The inequality ensures that $e < \phi(n)$ when an Approved algorithm is used to generate the primes $\underline{p,q}$.] If *requested_e* = 0 is supplied--the default value e=3 **should** be used.

3. The number k of output bits used for each iteration. The implementation **should** allow any multiple of 8 in the range $8 \le k \le \min\{ \lg(n) - 2*strength, \lg(n) - 2*\lg(n)/e \}$. If *requested_k* = 0 is specified, k **should** be selected as the *largest* multiple of 8 integer in the allowable range **and** within the range of bits currently known to be *hard* bits for the RSA problem. That value is lg(lg(n)), shown in Table 4. Thus, in all cases, the default value 8 will be used if *requested_k* = 0.

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

4. Set the size r of the seeds: $r = \lg(n) - k$.

5. Selection of the modulus n. The application may request a private modulus, or it may use the default modulus of the appropriate size as given in Annex A.2. The implementation **shall** permit either, based on the value of *use_random_primes*.

If *use_random_primes* = 1, two primes p and q of size ½lg(n) bits , having entropy at least *min_entropy*, and satisfying gcd (e, (p-1)(q-1)) = 1 **shall** be generated, using an approved algorithm . A suitable algorithm can be found in ANSI X9.31-1997, Annex B. An implementation **shall** use strong primes as defined in that document: each of p-1, p+1, q-1, q+1 must have a large prime factor of at least *strength* bits. [Comment: Any Approved algorithm will generate a modulus of size lg(n) bits using strong primes of size ½ lg(n) bits, and will allow the exponent e to be specified beforehand.]

The difficulty of the RSA problem relies on the secrecy of the primes p and q comprising the modulus. Whenever private primes are generated, the implementation **shall** clear memory of those values prior to leaving the instantiation routine. Only the modulus n **shall** be kept in the internal *state*.

If *use_random_primes* = 0 (*Use_random_primes*) the appropriate modulus from Annex A.2. **shall** be used. These modulii have been generated using strong primes of the form $p = 2*p_1 + 1$, $q = 2*q_1 + 1$, where p_1 and q_1 are themselves prime. In addition, p+1 and q+1 each have the required large prime factor. [Comment: This choice of strong primes essentially guarantees that any odd exponent e in the allowable range that might be requested will be relatively prime to $\phi(n)$.]

10.3.3.3.4 Instantiation of MS_DRBG (...)

The following process or its equivalent **shall** be used to instantiate the **MS_DRBG (...)** process. Let **Hash (...)** be an Approved hash function for the security strengths to be supported. If the DRBG will be used for multiple security strengths, and only a single hash function will be available, that hash function **shall** be suitable for all supported security strengths (see Table 4 and SP 800-57).

Instantiate_MS_DRBG (...):

Input: integer (*requested_strength*, *prediction_resistance_flag*, *personalization_string*, *use_random_primes*, *requested_e* , *requested_k* ,*reseed_interval*, *mode*).

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

Process:

```
1. If (requested_strength > the maximum security strength that can be provided by
   the implementation (see Table 4)), then Return ("Invalid requested_strength",
   Invalid_state_pointer).

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

                                   Comment: Find an empty state in the state
                                   space for the instantiation.

3. (status, state_pointer) = Find_state_space (mode).

4. If (status ≠ "Success"), Return (status, Invalid_state_pointer).
```

```
                                        Comment: Determine modulus size lg(n)
                                        appropriate for the requested strength using
                                        Table 4.
5. If (requested_strength $\le 80$) then {strength = 80, lg(n) = 1024}
   Else if (requested_strength $\le 112$) then {strength = 112, lg(n) = 2048}
      Else if (requested_strength $\le 128$) then {strength = 128, lg(n) = 3072}
         Else if (requested_strength $\le$ 192) then {strength = 192 , lg(n) = 7680}
            Else if (requested_strength $\le 256$) {strength = 256, lg(n) =
               15360}
               Else Return ("Invalid requested_strength",
                  Invalid_state_pointer).
6. If (requested_e = 0), then e = 3     Comment: Select the exponent size e. The
                                        default size is e=3.
   Else                                 Comment: Check the bounds. e must be at
                                        least 3.
      6.1 If ($e < 3$) Return ("Invalid requested_e", Invalid_state_pointer).
                                        Comment: e will need to be less than $\phi(n)$.
      6.2 If ($e \ge 2^{\lg(n)-1} - 2*2^{½\lg(n)}$), then Return ("Invalid requested_e",
          Invalid_state_pointer).
                                        Comment: e will need to be relatively prime
                                        to $\phi(n)$ , hence odd
      6.3 If (e is even) Return ("Invalid requested_e" , Invalid_state_pointer).
7. If (requested_k = 0 ), then          Comment : Select the output length k. The
                                        MS_DRBG (...) uses the least significant k
                                        bits of $y_i = si \| z_i$ on each iteration . The
                                        default size is to use the largest possible.
      7.1  k = min { $\lfloor \lg(n) - 2*strength \rfloor$ , $\lfloor \lg(n) * (1 - 2/e) \rfloor$.
                                        Comment: $3 \le e < 2^{\lg(n)-1} - 2*2^{½\lg(n)} \Rightarrow$
                                        $8 \le \lg(n) * 2/3 \le \lfloor \lg(n) * (1 - 2/e) \rfloor \le$
                                        $\lg(n) - 1$.
                                        Comment: Round down to a multiple of 8.
      7.2  $k = 8 * \lfloor k / 8 \rfloor$.
   Else                                 Comment: Check the bounds.
      7.3  k = requested_k.
      7.4  If ($k < 1$), then Return ("Inappropriate value for requested_k",
           Invalid_state_pointer).
```

```
        7.5 If ($k >$ min { $\lfloor \lg(n) - 2*strength \rfloor$, $\lfloor \lg(n) * (1 - 2/e) \rfloor$ } ), then
            Return ("Inappropriate value for requested_k ",
            Invalid_state_pointer).

        7.6 If ($k$ is not a multiple of 8), then Return ("Inappropriate value for
            requested_k", Invalid_state_pointer)

    8. $r = \lg(n) - k$                     Comment: Set the size of the seeds; $r \geq$
                                            $2*strength$.

                                            Comment: Select the modulus $n$.
                                            use_random_primes determines whether the
                                            default values are used or a private modulus is
                                            generated.

    9. If (use_random_primes = Random_primes_not_required) then

         Set $n$ based on the size $\lg(n)$  from the list in Annex A.2.

       Else                                 Comment: Use an approved function to
                                            generate a random modulus $n$ of the
                                            appropriate size, having strong primes as
                                            factors, and for which gcd ($\phi(n)$ ,$e$) = 1.

         If (Get_random_modulus ($\lg(n)$, $e$, $n$)) $\neq$ "Success "), then Return
         ("Failed to  produce an appropriate modulus", Invalid_state_pointer).
                                            Comment: Set the automatic reseeding
                                            interval. Automatic reseeding will not occur
                                            if a negative value for reseed_interval is
                                            provided.

    10. If (reseed_interval = 0), then reseed_interval = 50,000.

                                            Comment: Request entropy_input with the
                                            desired entropy and length $r$:

    11. min_entropy = max (128, strength).

    12. min_length = max_length = $r$

    13. (status, entropy_input) = Get_entropy (min_entropy, min_length, max_length,
        mode).

    14. If (status $\neq$ "Success"), then Return ("Failure indication returned by the
        entropy source", Invalid_state_pointer).

    15. seed_material = entropy_input || personalization_string.

    16. $S$ = Hash_df (seed_material, $r$).   Comment: Ensure that the entropy is
                                            distributed throughout the bits, and S is r bits
                                            in length.
```

Comment [ebb24]: Page: 29
Need to define/discuss this function.

Comment [ebb25]: Page: 29
Since step 16 will hash the entropy_input and personalization string down to *r* bits, a larger *max_length* should be allowed to allow for the case where full entropy input is not available.

Comment: Perform a one-way function on the seed material for later comparison.

17. *transformed_entropy_input* = **Hash** (*entropy_input*).

18. *reseed_counter* = 0.

Comment: *reseed_counter* will be incremented every *k* bits.

Comment: Store all values in *state* .

19. *state*(*state_pointer*) = {*reseed_counter*, *reseed_interval*, *S*, *n*, *e*, *r*, *k*, *strength*, *min_entropy*, *prediction_resistance_flag*, *transformed_entropy_input*}.

Comment [ebb26]: Page: 30
This could be omitted, since it can be calculated from the strength.

20. **Return** ("Success ").

10.3.3.3.5 Reseeding of a MS_DRBG (...) Instantiation

The following process or its equivalent **shall** be used to reseed the **MS_DRBG (...)** process, **after** it has been instantiated.

Reseed_MS_DRBG (...):

Input: integer *state_pointer,* string *additional_input_string,* integer *mode*).

Output: string *status.*

Process:

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

Comment: Get the required *state* values for the indicated *state_pointer.*

2. *min_entropy* = *state*(*state_pointer*).*min_entropy*, *S* = *state*(*state_pointer*).*S*, *r* = *state*(*state_pointer*).*r*, *old_transformed_entropy_input* = *state*(*state_pointer*).*transformed_entropy_input*.

Comment [ebb27]: Page: 30
This could be removed if we decide to calculate it below.

Comment: Request new *entropy_input.*

3. *min_entropy* = **max** (128, *strength*).

Comment [ebb28]: Page: 30
This could be omitted if you decide to keep min_entropy in the state.

4. *min_length* = *max_length* = *r*.

Comment [ebb29]: Page: 30
Since step 10 will hash the entropy_input and personalization string down to *r* bits, a larger *max_length* should be allowed to allow for the case where full entropy input is not available.

5. (*status*, *entropy_input*) = **Get_entropy** (*min_entropy*, *min_length*, *max_length*, *mode*).

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

Comment: Perform a one-way function on the seed material for comparison.

7. *transformed_entropy_input* = **Hash** (*entropy_input*).

   Comment : Check for a viable *entropy_input* source.

8. If (*transformed_entropy_input* = *old_transformed_entropy_input*), then

   If (*mode* = *Normal_operation*), then **Abort_to_error_state** ("*Entropy_input* source failure")

   Else Return ("*Entropy_input* source failure").

   Comment: Combine new *entropy_input* with the old state and any *additional_input.*

9. *seed_material* = *S* || *entropy_input* || *additional_input_string.*

10. *S* = **Hash_df** (*seed_material*, *r*).

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

11. *state(state_pointer).S* = *new_S.*

12. *state(state_pointer).transformed_entropy_input* = *old_transformed_entropy_input.*

13. *state(state_pointer).reseed_counter* = 0.

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

10.3.3.3.6 Generating Pseudorandom Bits Using MS_DRBG (...)

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

MS_DRBG (...):

Input: integer (*state_pointer*, *requested_strength, requested_no_of_bits, additional_input_string, prediction_resistance_request, mode*).

Output: string *status* , bitstring *pseudorandom_bits*

Process:

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

   Comment: Get the appropriate *state* for the indicated *state_pointer.*

2. *S* = *state(state_pointer).S*, *n* = *state(state_pointer).n*, *e* = *state(state_pointer).e*, *k* = *state(state_pointer).k*, *r* = *state(state_pointer).r*, *strength* = *state(state_pointer).strength*, *reseed_counter* =

```
state(state_pointer).reseed_counter, reseed_interval =
state(state_pointer).reseed_interval, prediction_resistance_flag =
state(state_pointer). prediction_resistance_flag.

                                        Comment: Check that the requested strength
                                        is not larger than that provide by this
                                        instantiation.

3. If (requested_strength > strength), then Return ("Invalid requested_strength",
   Null).

4. If ((prediction_resistance_request = Provide_prediction_resistance) and
   (prediction_resistance_flag = No_prediction_resistance)), then Return
   ("Prediction resistance capability not instantiated", Null).

                                        Comment: Check for supplied additional
                                        input. This will be added to the state on the
                                        first iteration only.

5. If (additional_input_string = Null) then  additional_input = 0

                                        Comment:  additional_input set to r zeroes.

   Else additional_input = Hash_df (pad8 (additional_input_string), r).

                                        Comment: Hash to r bits.

                                        Comment: If a prediction resistance request
                                        has been made, instill new entropy with a call
                                        to  reseed the MS_DRBG(...).  This is only
                                        allowed if prediction resistance was requested
                                        at instantiation. The reseedin process resets
                                        reseed_counter to 0.

6. If (prediction_resistance_request = Provide_prediction_resistance) then

   6.1 status = Reseed_MS_DRBG_Instantiation (state_pointer, Null, mode).

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

                                        Comment:  Produce requested_no_of_bits , k
                                        at a time.

7. temp = the Null string.

8. i = 0.

                                        Comment: Determine if reseeding is required.
                                        Reseed() resets reseed_counter to 0.

9. If ((reseed_interval > 0) and (reseed_counter = reseed_interval)), then

   9.1   status = Reseed_MS_DRBG_Instantiation (state_pointer, Null, mode).
```

```
    9.2 If ($status \neq$ "Success"), then Return ($status, Null$).
10. $s = S \oplus additional\_input$.          Comment: $s$ is to be interpreted as an $r$-bit
                                               unsigned integer.
11. $S = [\ (s^e \bmod n) / 2^k\ ]$.           Comment: $S$ is an $r$-bit number.
12. $R = (s^e \bmod n) \bmod 2^k$.             Comment: $R$ is a $k$-bit number.
13. $temp = temp \,||\, R$.
14. $additional\_input = 0$.                   Comment: $r$ zeroes; additional_input_string
                                               is added only on the first iteration.
15. $i = i + 1$.
16. $reseed\_counter = reseed\_counter + 1$.
17. If (len ($temp$) < $requested\_no\_of\_bits$), then go to step 9.
18. $pseudorandom\_bits$ = Truncate ($temp, i \times k, requested\_no\_of\_bits$).
                                               Comment: Update the changed values
                                               in the state.
19. $state.S = S$.
20. $state.reseed\_counter = reseed\_counter$.
21. Return ("Success", $pseudorandom\_bits$).
```

Comment [ebb30]: Page: 33
This was done so that there is no confusion with concatenation.

10.3.3.3.8 Removing an MS_DRBG (...) Instantiation

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

Uninstantiate_MS_DRBG (...):

Input: integer *state_pointer*.

Output: string *status*.

Process:

```
1. If ($state\_pointer > max\_no\_of\_states$), then Return ("Invalid state_pointer").
2. $state(state\_pointer) = \{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, Null\}$.
3. Return ("Success").
```

10.3.3.3.9 Self Testing of the Dual_EC_DRBG (...)

Self testing **shall** be performed on the **MS_DRBG (...)** processes contained within a DRBG boundary using the specifications in Section 9.9.

[Differences to be determined].

10.3.3.3.10 Implementation Considerations

The **Get_entropy (...)** function is implementation dependent. Depending on the environment, the entropy source may be an approved NRBG which is gathering entropy in the background, or perhaps a hardware device specifically for this purpose. The implementation may pause while the requested entropy is gathered (if so documented); it **shall** return an error status if the requested entropy cannot be satisfied.

In deference to the defacto standard of using character arrays as inputs and outputs for hash implementations, **MS_DRBG(...)** pads bitstrings to byte boundaries, and requests its seed material to comply as well.

Comment [ebb31]: Page: 34
Reference ?

Comment [ebb32]: Page: 34
This subject weas handled differently for the hash_based DRBGs. We need to decide the best way to do this.

10.3.3.4 Generator Strength and Attributes

The size of the RSA modulus *n* is based on *strength*, which is selected from one of five security levels and is always at least *requested_strength*. The sizes have been chosen to comply with FIPS published standards.

Initial seeding is accomplished with a call to **Get_entropy (...)**, which returns a bitstring of a specified length and entropy. The **MS_DRBG (...)** specifies **max** (128, *strength*) bits of entropy.

10.3.3.5 Reseeding and Rekeying

The reseeding process is covered in Section 10.3.3.3.5. Automatic reseeding is done by default after each 50,000 blocks of *k* bits of random are output. The frequency may be changed by providing a positive value for *reseed_interval*, or the feature may be disabled by setting *reseed_interval* < 0 at instantiation. Alternatively, or in addition, a call to **Reseed_MS_DRBG_Instantiation (...)** can be made at any time.

The **MS_DRBG** (...) is not keyed per se; however, the *additional_input* and *personalization_string* features may be used to effect keying, if desired.

Comment [ebb33]: Page: 34
Need to decide if we need this.