X9.82 DRBG ALGORITHMS

HMAC_DRBG:

10.1.2.2.2 The Update Function (Update)

The **Update** function updates the internal state of **HMAC_DRBG** using the *provided_data*. Note that for this DRBG, the **Update** function also serves as a derivation function for the instantiate and reseed functions.

Let **HMAC** be the keyed hash function specified in FIPS 198 using the hash function selected for the DRBG from Table 2 in Section 10.1.1.

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

Input:

1. *provided_data*: The data to be used.
2. *K*: The current value of *Key*.
3. *V*: The current value of *V*.

Output:

1. *K*: The new value for *Key*.
2. *V*: The new value for *V*.

Process:

1. *K* = **HMAC** (*K*, *V* || 0x00 || *provided_data*).
2. *V* = **HMAC** (*K*, *V*).
3. If (*provided_data* = *Null*), then return *K* and *V*.
4. *K* = **HMAC** (*K*, *V* || 0x01 || *provided_data*).
5. *V* = **HMAC** (*K*, *V*).
6. Return *K* and *V*.

10.1.2.2.3 Instantiation of HMAC_DRBG

Notes for the instantiate function:

The instantiation of **HMAC_DRBG** requires a call to the instantiate function specified in Section 9.2: step 9 of that function calls the instantiate algorithm specified in this section. For this DRBG, step 5 **should** be omitted. The values of *highest_supported_security_strength* and *min_length* are provided in Table 2 of Section 10.1.1. The contents of the internal state are provided in Section 10.1.2.2.1.

The instantiate algorithm:

Let **Update** be the function specified in Section 10.1.2.2.2. The output block length (*outlen*) is provided in Table 2 of Section 10.1.1.

The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG (see step 9 of Section 9.2):

Input:

1. *entropy_input*: The string of bits obtained from the entropy input source.
2. *nonce*: A string of bits as specified in Section 8.4.2.
3. *personalization_string*: The personalization string received from the consuming application. If a *personalization_string* will never be used, then step 1 may be modified to remove the *personalization_string*.

Output:

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

Process:

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

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

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

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

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

5. *reseed_counter* = 1.
6. Return *V*, *Key* and *reseed_counter* as the initial *working_state*.

10.1.2.2.4 Reseeding an HMAC_DRBG Instantiation

Notes for the reseed function:

The reseeding of an **HMAC_DRBG** instantiation requires a call to the reseed function specified in Section 9.3; step 5 of that function calls the reseed algorithm specified in this section. The values for *min_length* are provided in Table 2 of Section 10.1.1.

The reseed algorithm:

Let **Update** be the function specified in Section 10.1.2.2.2. The following process or its equivalent **shall** be used as the reseed algorithm for this DRBG (see step 5 of Section 9.3):

Input:

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

2. *entropy_input*: The string of bits obtained from the entropy input source.
3. *additional_input*: The additional input string received from the consuming application. If *additional_input* will never be used, then step 1 may be modified to remove the *additional_input*.

Output:

1. *working_state*: The new values for *V*, *Key* and *reseed_counter*.

Process:

1. *seed_material* = *entropy_input* || *additional_input*.
2. (*Key*, *V*) = **Update** (*seed_material*, *Key_old*, *V_old*).
3. *reseed_counter* = 1.
4. Return *V*, *Key* and *reseed_counter* as the new *working_state*.

10.1.2.2.5 Generating Pseudorandom Bits Using HMAC_DRBG

Notes for the generate function:

The generation of pseudorandom bits using an **HMAC_DRBG** instantiation requires a call to the generate function specified in Section 9.4: step 8 of that function calls the generate algorithm specified in this section. The values for *max_number_of_bits_per_request* and *outlen* are provided in Table 2 of Section 10.1.1.

The generate algorithm :

Let **HMAC** be the keyed hash function specified in FIPS 198 using the hash function selected for the DRBG. The value for *reseed_interval* is defined in Table 2 of Section 10.1.1.

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

Input:

1. *working_state*: The current values for *V*, *Key* and *reseed_counter* (see Section 10.1.2.2.1).
2. *requested_number_of_bits*: The number of pseudorandom bits to be returned to the generate function.
3. *additional_input*: The additional input string received from the consuming application. If an implementation will never use *additional_input*, then step 3 may be omitted. If an implementation does not include the *additional_input* parameter as one of the calling parameters, or if the implementation allows *additional_input*, but a given request does not provide any *additional_input*, then a *Null* string **shall** be used as the *additional_input* in step 7.

Output:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS,** an **ERROR** or indicate that a reseed is required before the requested pseudorandom bits can be generated.
2. *returned_bits*: The pseudorandom bits to be returned to the generate function.
3. *working_state*: The new values for *V*, *Key* and *reseed_counter*.

Process:

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

   Comment: Save the last ouput block for comparison with the new output block.

2. *V_old* = *V*. *Key_old* = *Key*.
3. If *additional_input* ≠ *Null*, then (*Key*, *V*) = **Update** (*additional_input*, *Key*, *V*).
4. *temp* = *Null*.
5. While (**len** (*temp*) < *requested_number_of_bits*) do:

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

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

   ~~5.2 If (*V* = *V_old*), then return an **ERROR**.~~

   ~~5.3 *V_old* = *V*.~~

   5.4 *temp* = *temp* || *V*.

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

[Insert] If ((*Key* = *Key_old*) and (*V_old* = *V*)), then return an **ERROR**.

8. *reseed_counter* = *reseed_counter* + 1.
9. Return **SUCCESS**, *returned_bits*, and the new values of *Key*, *V* and *reseed_counter* as the *working_state*).

CTR_DRBG :

10.2.2.2.1 CTR_DRBG Internal State

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

1. The *working_state*:
   a. The value *V* of *outlen* bits, which is updated each time another *outlen* bits of output are produced (see Table 3 in Section 10.2.2.1).
   b. The *keylen*-bit *Key*, which is updated whenever a predetermined number of output blocks are generated.
   ~~c. The *previous_output_block*; this is required to perform a continuous test on the output from the generate function.~~
   d. A counter (*reseed_counter*) that indicates the number of requests for pseudorandom bits since instantiation or reseeding.
2. Administrative information:
   a. The *security_strength* of the DRBG instantiation.
   b. A *prediction_resistance_flag* that indicates whether or not a prediction resistance capability is required for the DRBG.

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

10.2.2.2.2 The Update Function (Update)

The **Update** function updates the internal state of the **CTR_DRBG** using the *provided_data*. The values for *outlen*, *keylen* and *seedlen* are provided in Table 3 of Section 10.2.2.1. The block cipher operation in step 2.2 uses the selected block cipher algorithm (also see Section 9.1).

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

Input:

1. *provided_data*: The data to be used. This must be exactly *seedlen* bits in length; this length is guaranteed by the construction of the *provided_data* in the instantiate, reseed and generate functions.
2. *Key*: The current value of *Key*.
3. *V*: The current value of *V*.

Output:

1. *K*: The new value for *Key*.
2. *V*: The new value for *V*.

Process:

1. *temp* = *Null*.

2. While (**len** (*temp*) < *seedlen*) do

   2.1 $V = (V + 1) \bmod 2^{outlen}$.

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

   2.3 *temp* = *temp* || *ouput_block*.

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

4 *temp* = *temp* ⊕ *provided_data*.

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

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

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

10.2.2.2.3 Instantiation of CTR_DRBG

Notes for the instantiate function:

The instantiation of **CTR_DRBG** requires a call to the instantiate function specified in Section 9.2; step 9 of that function calls the instantiate algorithm specified in this section. For this DRBG, step 5 **should** be omitted. The values of *highest_supported_security_strength* and *min_length* are provided in Table 3 of Section 10.2.2.1. The contents of the internal state are provided in Section 10.2.2.2.1.

The instantiate algorithm:

Let **Update** be the function specified in Section 10.2.2.2.2. The output block length (*outlen*), key length (*keylen*), seed length (*seedlen*) and *security_strengths* for the block cipher algorithms are provided in Table 3 of Section 10.2.2.1.

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

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

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

Input:

1. *entropy_input*: The string of bits obtained from the entropy input source.
2. *nonce*: A string of bits as specified in Section 8.4.2; this string **shall not** be present unless a derivation function is used.
3. *personalization_string*: The personalization string received from the consuming application.

Output:

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

Process:

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

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

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

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

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

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

   1.5 *seed_material* = *entropy_input* ⊕ *personalization_string*.

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

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

6. *previous_output_block* = **Block_Encrypt** (*Key*, *V*).
7. *zeros* = $0^{seedlen}$. — Comment: Produce a string of *seedlen* zeros.
8. (*Key*, *V*) = **Update** (*zeros*, *Key*, *V*).
9. Return *V*, *Key*, *previous_output_block* and *reseed_counter* as the *working_state*.

Implementation notes:

1. If a *personalization_string* will never be provided from the instantiate function and a derivation function will be used, then step 1.1 becomes:

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

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

   seed_material = *entropy_input*.

   That is, steps 1.3 – 1.5 collapse into the above step.

10.2.2.2.4 Reseeding a CTR_DRBG Instantiation

Notes for the reseed function:

The reseeding of a **CTR_DRBG** instantiation requires a call to the reseed function specified in Section 9.3; step 5 of that function calls the reseed algorithm specified in this section. The values for *min _length* are provided in Table 3 of Section 10.2.2.1.

The reseed algorithm:

Let **Update** be the function specified in Section 10.2.2.2.2. The seed length (*seedlen*) is provided in Table 3 of Section 10.2.2.1.

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

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

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

Input:

1. *working_state*: The current values for *V*, *Key, previous_output_block* and *reseed_counter* (see Section 10.2.2.2.1).
2. *entropy_input*: The string of bits obtained from the entropy input source.
3. *additional_input*: The additional input string received from the consuming application.

Output:

1. *working_state*: The new values for *V*, *Key, previous_output_block* and *reseed_counter.*

Process:

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

   1.1 *seed_material* = *entropy_input* || *additional_input.*

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

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

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

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

1.5 *seed_material* = *entropy_input* ⊕ *additional_input*.

2. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).
3. *reseed_counter* = 1.
4. **Return** *V*, *Key*, *previous_output_block* and *reseed_counter* as the *working_state*.

Implementation notes:

1. If *additional_input* will never be provided from the reseed function and a derivation function will be used, then step 1.1 becomes:

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

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

   seed_material = *entropy_input*.

   That is, steps 1.3 – 1.5 collapse into the above step.

10.2.2.2.5 Generating Pseudorandom Bits Using CTR_DRBG

Notes for the generate function:

The generation of pseudorandom bits using a **CTR_DRBG** instantiation requires a call to the generate function specified in Section 9.4, step 8 of that function calls the generate algorithm specified in this section. The values for *max_number_of_bits_per_request* and *outlen* are provided in Table 3 of Section 10.2.2.1. If the derivation function is not used, then the maximum allowed length of *additional_input* = *seedlen*.

Let **Update** be the function specified in Section 10.2.2.2.2. The seed length (*seedlen*) and the value of *reseed_interval* are provided in Table 3 of Section 10.2.2.1. Step 5.2 below uses the selected block cipher algorithm. If a derivation function is not used for a DRBG implementation, then step 3.2 **shall** be omitted.

If a block cipher derivation function is to be used, then the **Block_Cipher_df** specified in Section 9.6.3 **shall** be implemented using the chosen block cipher algorithm and key size; in this case, step 3.2 below **shall** be included.

If full entropy is available whenever entropy input is required, and a block cipher derivation function is not to be used, then step 3.2 below **shall not** be used.

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

Input:

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

2. *requested_number_of_bits*: The number of pseudorandom bits to be returned to the generate function.
3. *additional_input*: The additional input string received from the consuming application. If *additional_input* will never be provided, then step 3 may be omitted.

Output:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS,** an **ERROR** or indicate that a reseed is required before the requested pseudorandom bits can be generated.
2. *returned_bits*: The pseudorandom bits returned to the generate function.
3. *working_state*: The new values for *V*, *Key*, *previous_output_block* and *reseed_counter*.

Process:

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

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

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

   Comment: If a block cipher derivation function is used:

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

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

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

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

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

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

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

5.3 If (*output_block* = *previous_output_block*), then return an **ERROR**.

5.4 *previous_output_block* = *output_block*.

5.5 *temp* = *temp* || *ouput_block*.

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

Comment: Update for backtracking resistance.

7. *zeros* = $0^{seedlen}$.

Comment: Produce a string of *seedlen* zeros.

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

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

10. Return **SUCCESS** and *returned_bits*; also return *Key*, *V*, *previous_output_block* and *reseed_counter* as the new *working_state*.

10.3 Deterministic RBG Based on Number Theoretic Problems

10.3.1 Discussion

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

10.3.2 Dual Elliptic Curve Deterministic RBG (Dual_EC_DRBG)

10.3.2.1 Discussion

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

Dual_EC_DRBG uses a seed that is m bits in length (i.e., *seedlen* = m) to initiate the generation of *outlen*-bit pseudorandom strings by performing scalar multiplications on two points in an elliptic curve group, where the curve is defined over a field approximately 2^m in size. For all of the NIST curves given in this Standard for the DRBG, $m \geq 224$. Figure 11 depicts the **Dual_EC_DRBG**.

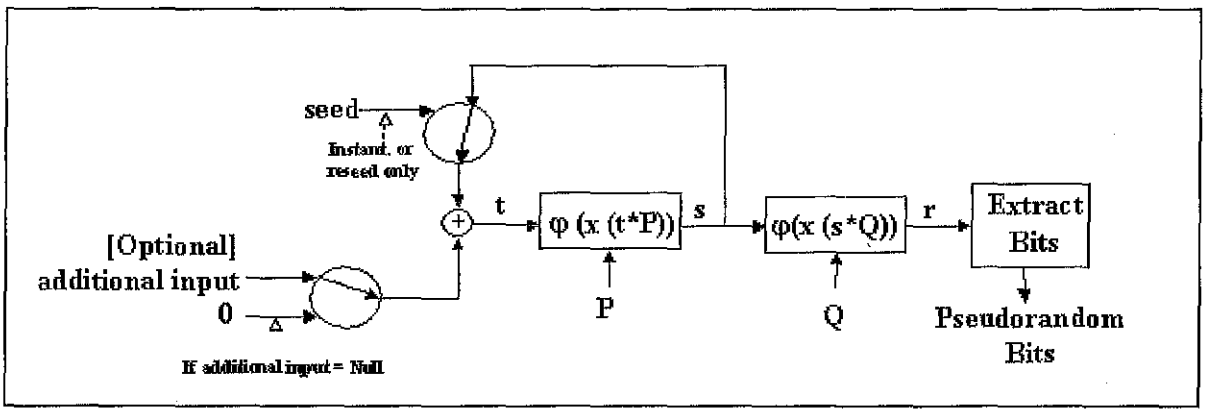

Figure 11: Dual_EC_DRBG

The instantiation of this DRBG requires the selection of an appropriate elliptic curve and curve points specified in Annex A.1 for the desired security strength. Requirements for the *seed* are provided in Section 8.4.2.

Backtracking resistance is inherent in the algorithm, even if the internal state is compromised. As shown in Figure 12, **Dual_EC_DRBG** generates a *seedlen*-bit number for each step $i = 1,2,3,\ldots$, as follows:

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

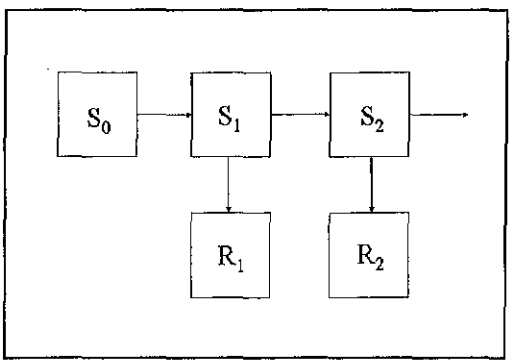

Figure 12: Dual_EC_DRBG (...)

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

Each arrow in the figure represents an Elliptic Curve scalar multiplication operation, followed by the extraction of the x coordinate for the resulting point and for the random output R_i, and by truncation to produce the output (formal definitions for φ and x are given in Section 10.3.2.2.4). Following a line in the direction of the arrow is the normal operation; inverting the direction implies the ability to solve the ECDLP for that specific curve. An adversary's ability to invert an arrow in the figure implies that the adversary has solved the ECDLP for that specific elliptic curve. Backtracking resistence is built into the design, as knowledge of S_1 does not allow an adversary to determine S_0 (and so forth) unless the adversary is able to solve the ECDLP for that specific curve. In addition, knowledge of R_1 does not allow an adversary to determine S_1 (and so forth) unless the adversary is able to solve the ECDLP for that specific curve.

Table 4 specifies the values that **shall** be used for the envelope and algorithm for each curve. Complete specifications for each curve are provided in Annex A.1. Note that all curves except the P-224 curve can be instantiated at a security strength lower than its highest possible security strength. For example, the highest security strength that can be supported by curve P-384 is 192 bits; however, this curve can alternatively be instantiated to support only the 112 or 128-bit security strengths).

Table 4: Definitions for the Dual_EC_DRBG

	P-224	**P-256**	**P-384**	**P-521**
Supported security strengths	See SP 800-57			
highest_supported_ security_strength	See SP 800-57			
Output block length (*max_outlen* = largest multiple of 8 less than *seedlen* - (13 + $\log_2$ (the cofactor))	208	240	368	504
Required minimum entropy for instantiate and reseed	*security_strength*			
Minimum entropy input length (*min_length* = 8 × $\lceil$ *seedlen*/8 $\rceil$)	224	256	384	528
Maximum entropy input length (*max_length*)	$\leq 2^{13}$ bits			
Maximum personalization string length (*max_personalization_string_length*)	$\leq 2^{13}$ bits			
Supported security strengths	See SP 800-57			
Seed length (*seedlen* = *m*)	224	256	384	521

	P-224	P-256	P-384	P-521
Appropriate hash functions	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512		SHA-224, SHA-256, SHA-384, SHA-512	SHA-256, SHA-384, SHA-512
max_number_of_bits_per_request	*max_outlen* × *reseed_interval*			
Number of blocks between reseeding (*reseed_interval*)	$\leq 2^{32}$ blocks			

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

10.3.2.2 Specifications

10.3.2.2.1 Dual_EC_DRBG Internal State

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

1. The *working_state*:
   a. A value (*s*) that determines the current position on the curve.
   b. The elliptic curve domain parameters (*seedlen*, *p*, *a*, *b*, *n*), where *seedlen* is the length of the seed ; *a* and *b* are two field elements that define the equation of the curve, and *n* is the order of the point *G*. If only one curve will be used by an implementation, these parameters need not be present in the *working_state*.
   c. Two points *P* and *Q* on the curve; the generating point *G* specified in FIPS 186-3 for the chosen curve will be used as *P*. If only one curve will be used by an implementation, these points need not be present in the *working_state*.
   d. *r_old*, the previous output block.
   e. A counter (*block_counter*) that indicates the number of blocks of random produced by the **Dual_EC_DRBG** since the initial seeding or the previous reseeding.
2. Administrative information:
   a. The *security_strength* provided by the instance of the DRBG,
   b. A *prediction_resistance_flag* that indicates whether prediction resistance is required by the DRBG.

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

10.3.2.2.2 Instantiation of Dual_EC_DRBG

Notes for the instantiate function:

The instantiation of **Dual_EC_DRBG** requires a call to the instantiate function specified in Section 9.2; step 9 of that function calls the instantiate algorithm in this section.

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

5. Using the *security_strength* and Table 4 in Section 10.3.2.1, select the smallest available curve that has a security strength ≥ *security_strength*.

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

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

The values for *highest_supported_security_strength* and *min_length* are determined by the selected curve (see Table 4 in Section 10.3.2.1).

The instantiate algorithm :

Let **Hash_df** be the hash derivation function specified in Section 9.6.2 using an appropriate hash function from Table 4 in Section 10.3.2.1. Let *seedlen* be the appropriate value from Table 4.

The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG (see step 9 of Section 9.2):

Input:

1. *entropy_input*: The string of bits obtained from the entropy input source.
2. *nonce*: A string of bits as specified in Section 8.4.2.
3. *personalization_string*: The personalization string received from the consuming application.

Output:

1. *s*: The initial secret value for the *working_state*.
2. *r_old*: The initial output block (which will not be used).
3. *block_counter*: The initialized block counter for reseeding.

Process:

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

   Comment: Use a hash function to ensure that the entropy is distributed throughout the bits, and *s* is *m* (i.e., *seedlen*) bits in length.

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

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

3. *r_old* = φ(*x*(*s* * *Q*)). Comment: *r* is a *seedlen*-bit number.
4. *block_counter* = 0.
5. Return *s*, *r_old* and *block_counter* for the *working_state*.

10.3.2.2.3 Reseeding of a Dual_EC_DRBG Instantiation

Notes for the reseed function:

The reseed of **Dual_EC_DRBG** requires a call to the reseed function specified in Section 9.3; step 5 of that function calls the reseed algorithm in this section. The values for *min _length* are provided in Table 4 of Section 10.3.2.1.

The reseed algorithm :

Let **Hash_df** be the hash derivation function specified in Section 9.6.2 using an appropriate hash function from Table 4 in Section 10.3.2.1.

The following process or its equivalent **shall** be used to reseed the **Dual_EC_DRBG** process after it has been instantiated (see step 4 in Section 9.3):

Input:

1. *s*: The current value of the secret parameter in the *working_state*.
2. *entropy_input*: The string of bits obtained from the entropy input source.
3. *additional_input*: The additional input string received from the consuming application.

Output:

1. *s*: The new value of the secret parameter in the *working_state*.
2. *block_counter*: The re-initialized block counter for reseeding.

Process:

Comment: **pad8** returns a copy of *s* padded on the right with binary 0's, if necessary, to a multiple of 8.

1. *seed_material* = **pad8** (*s*) || *entropy_input* || *additional_input_string*.
2. *s* = **Hash_df** (*seed_material*, *seedlen*).
3. *block_counter* = 0.
4. Return *s* and *block_counter* for the new *working_state*.

Implementation notes:

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

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

10.3.2.2.4 Generating Pseudorandom Bits Using Dual_EC_DRBG

Notes for the generate function:

The generation of pseudorandom bits using a **Dual_EC_DRBG** instantiation requires a call to the generate function specified in Section 9.4; step 8 of that function calls the generate algorithm specified in this section. The values for *max_number_of_bits_per_request* and *max_outlen* are provided in Table 4 of Section 10.3.2.1. *outlen* is the number of pseudorandom bits taken from each *x*-coordinate as the **Dual_EC_DRBG** steps. For performance reasons, the value of *outlen* should be set to the maximum value as provided in Table 5. However, an implementation **may** set *outlen* to any multiple of 8 bits less than or equal to *max_outlen*. The bits that become the **Dual_EC_DRBG** output are always the rightmost bits, i.e., the least significant bits of the *x*-coordinates.

The generate algorithm:

Let **Hash_df** be the hash derivation function specified in Section 9.6.2 using an appropriate hash function from Table 4 in Section 10.3.2.1. The value of *reseed_interval* is also provided in Table 4.

The following are used by the generate algorithm:

a. **pad8** (bitstring) returns a copy of the *bitstring* padded on the right with binary 0's, if necessary, to a multiple of 8.

b. **Truncate** (*bitstring, in_len, out_len*) inputs a *bitstring* of *in_len* bits, returning a string consisting of the leftmost *out_len* bits of *bitstring*. If *in_len* < *out_len*, the *bitstring* is padded on the right with (*out_len* - *in_len*) zeroes, and the result is returned.

c. $x(A)$ is the *x*-coordinate of the point A on the curve, given in affine coordinates. An implementation may choose to represent points internally using other coordinate systems; for instance, when efficiency is a primary concern. In this case, a point **shall** be translated back to affine coordinates before *x*() is applied.

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

The precise definition of $\varphi(x)$ used in steps 6 and 7 below depends on the field representation of the curve points. In keeping with the convention of FIPS 186-2, the following elements will be associated with each other (note that m = *seedlen*):

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

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

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

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

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

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

Input:

1. *working_state*: The current values for *s, seedlen, p, a, b, n, P, Q, r_old* and *reseed_counter* (see Section 10.1.3.2.1).
2. *requested_number_of_bits*: The number of pseudorandom bits to be returned to the generate function.
3. *additional_input*: The additional input string received from the consuming application.

Output:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS, ERROR** or an indication that a reseed is required before the requested pseudorandom bits can be generated.
2. *returned_bits*: The pseudorandom bits to be returned to the generate function.
3. *s*: The new value for the secret parameter in the *working_state*.
4. *r_old*: The last output block.
5. *block_counter*: The updated block counter for reseeding.

Process:

Comment: Check whether a reseed is required.

1. If $\left(block\_counter + \left\lceil \frac{requested\_number\_of\_bits}{outlen} \right\rceil \right) > reseed\_interval$, then return an indication that a reseed is required.

Comment: If *additional_input* is *Null*, set to *seedlen* zeroes; otherwise, **Hash_df** to *seedlen* bits.

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

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

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

3. *temp* = the *Null* string.
4. $i = 0$.

5. $t = s \oplus additional\_input$. Comment: t is to be interpreted as a *seedlen*-bit unsigned integer. To be precise, t should be reduced mod n; the operation * will effect this.

6. $s = \varphi(x(t * P))$. Comment: s is a *seedlen*-bit number.

7. $r = \varphi(x(s * Q))$. Comment: r is a *seedlen*-bit number.

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

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

9. $r\_old = r$.

10. $temp = temp \,||$ (**rightmost** *outlen* bits of r).

11. $additional\_input$=0 Comment: *seedlen* zeroes; *additional_input_string* is added only on the first iteration.

12. $block\_counter = block\_counter + 1$.

13. $i = i + 1$.

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

15 *returned_bits* = **Truncate** (*temp*, $i \times outlen$, *requested_number_of_bits*).

16. Return **SUCCESS**, *returned_bits,* and *s*, *r_old* and *block_counter* for the *working_state.*