I've provided the modified Dual_EC_DRBG section below. The gray highlight is used to remind me that I may have to modify that text, depending on what happens in the rest of the document, e.g., figure numbers change.

FIPS 140-2 (Section 4.9.2) has the following statements :

> *Continuous random number generator test.* If a cryptographic module employs Approved or non-Approved RNGs in an Approved mode of operation, the module shall perform the following continuous random number generator test on each RNG that tests for failure to a constant value.
>
> 1. If each call to a RNG produces blocks of n bits (where $n > 15$), the first n-bit block generated after power-up, initialization, or reset shall not be used, but shall be saved for comparison with the next n-bit block to be generated. Each subsequent generation of an n-bit block shall be compared with the previously generated block. The test shall fail if any two compared n-bit blocks are equal.
> 2. If each call to a RNG produces fewer than 16 bits, the first n bits generated after power-up, initialization, or reset (for some $n > 15$) shall not be used, but shall be saved for comparison with the next n generated bits. Each subsequent generation of n bits shall be compared with the previously generated n bits. The test fails if any two compared n-bit sequences are equal.

I'm assuming that case 1 applies to us. I tried to address this in the previous version, but goofed. I've made the text in the routines below red where I think I've addressed this situation. Does it look OK?

I've done the following:

a. Added a new value to the internal state (*r_old*) to hold the 1st/previous output block).

b. In the instantiate algorithm (10.3.2.2.2), I used the $\varphi(x(s * Q))$ function to generate the 1st value of r (that won't actually be output).

c. In the generate algorithm (10.3.2.2.4), the old and new values of r are compared. If they match, an error is returned. In the generate function (9.4), the error will cause all instantiations to be emptied, under the assumption that if subsequent values of r match, there is a dire problem somewhere, so disable the DRBG.

10.3 Deterministic RBGs Based on Number Theoretic Problems

10.3.1 Discussion

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

10.3.2 Dual Elliptic Curve Deterministic RBG (Dual_EC_DRBG)

10.3.2.1 Discussion

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

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

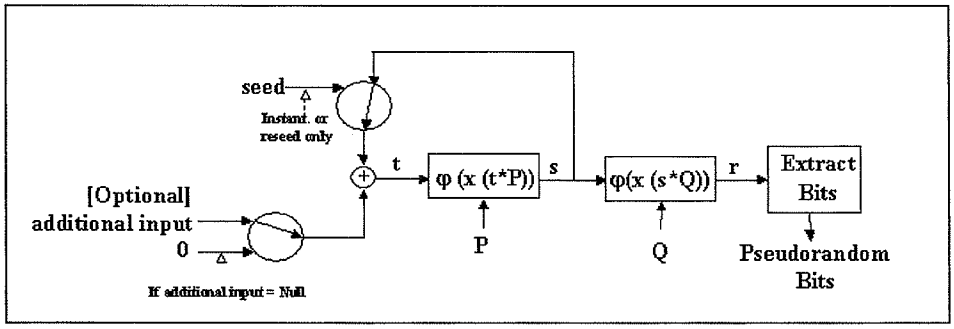

Figure 15: Dual_EC_DRBG

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

Backtracking resistance is inherent in the algorithm, even if the internal state is compromised. As shown in Figure 16, **Dual_EC_DRBG** generates a *seedlen*-bit number for each step $i = 1,2,3,\ldots$, as follows:

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

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

Each arrow in the figure represents an Elliptic Curve scalar multiplication operation, followed by the extraction of the x coordinate for the resulting point and for the random output R_i, and by truncation to produce the output. 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

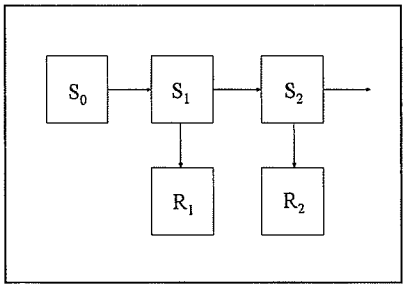

Figure 16: Dual_EC_DRBG (...) Backtracking Resistance

into the design, as knowledge of S_1 does not allow an adversary to determine S_0 (and so forth) unless the adversary is able to solve the ECDLP for that specific curve. In addition, knowledge of R_1 does not allow an adversary to determine S_1 (and so forth) unless the adversary is able to solve the ECDLP for that specific curve.

Table 5 specifies the values that **shall** be used for the envelope and algorithm for each curve. Complete specifications for each curve are provided in Annex A.1. Note that all curves except the first three can be instantiated at a security strength lower than its highest possible security strength. For example, the highest security strength that can be supported by curve P-384 is 192 bits; however, this curve can alternatively be instantiated to support only the 112 or 128-bit security strengths).

Table 5: Definitions for the Dual_EC_DRBG

	P-224	P-256	P-384	P-521
Supported security strengths	See SP 800-57			
highest_supported_ security_strength	See SP 800-57			
Output block length (*max_outlen* = largest multiple of 8 less than *seedlen* - (13 + $\log_2$ (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
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*			

Comment [ebb1]: Page: 78
Why can't this be min_entropy ?

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

10.3.2.2 Specifications

10.3.2.2.1 Dual_EC_DRBG Internal State and Other Specification Details

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

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

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

10.3.2.2.2 Instantiation of Dual_EC_DRBG

Notes for the instantiate function:

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

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

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

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

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

The values for *highest_supported_security_strength* and *min_length* are determined by the selected curve (see Table 5 in Section 10.3.2.1).

The instantiate algorithm :

Let **Hash_df** be the hash derivation function specified in Section 9.6.2 using an appropriate hash function from Table 5 in Section 10.3.2.1. Let *seedlen* be the appropriate value from Table 5.

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

Input:

1. *entropy_input*: The string of bits obtained from the entropy input source.
2. *nonce*: A string of bits as specified in Section 8.4.
3. *personalization_string*: The personalization string received from the consuming application.

Output:

1. *s*: The initial secret value for the *working_state*.
2. *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* = $\varphi(x(s * Q))$. Comment: *r* is a *seedlen*-bit number.

4. *block_counter* = 0.

5. Return *s*, *r_old* and *block_counter* for the *working_state*.

Comment [ebb2]: Page: 1
Is this suitable to use to compare withe the next value of *r* used by the generate function for continuous testing of the output blocks ?

Implementation notes:

If an implementation never uses a *personalization_string*, then steps 1 and 2 may be combined as follows :

s = **Hash_df** (*entropy_input*, *seedlen*).

10.3.2.2.3 Reseeding of a Dual_EC_DRBG Instantiation

Comment [ebb3]: Page: 82
Need to add steps to perform the « continuous » test.

Notes for the reseed function:

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

The reseed algorithm :

Let **Hash_df** be the hash derivation function specified in Section 9.6.2 using an appropriate hash function from Table 5 in Section 10.3.2.1.

The following process or its equivalent **shall** be used to reseed the **Dual_EC_DRBG** process after it has been instantiated (see step 5 in Section 9.3):

Input:

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

Output:

1. *status*: The status returned from the reseed function.
2. *s*: The new value of the secret parameter in the *working_state.*
3. *block_counter*: The re-initialized block counter for reseeding.

Process:

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

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

Implementation notes:

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

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

10.3.2.2.4 Generating Pseudorandom Bits Using Dual_EC_DRBG

Notes for the generate function:

The generation of pseudorandom bits using a **Dual_EC_DRBG** instantiation requires a call to the generate function specified in Section 9.4; step 8 of that function calls the generate algorithm specified in this section. The values for *max_number_of_bits_per_request* and *max_outlen* are provided in Table 5 of Section 10.3.2.1. *outlen* is the number of pseudorandom bits taken from each *x*-coordinate as

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

The generate algorithm:

Let **Hash_df** be the hash derivation function specified in Section 9.6.2 using an appropriate hash function from Table 5 in Section 10.3.2.1. The value of *reseed_interval* is also provided in Table 5.

The following are used by the generate algorithm:

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

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

c. $x(A)$ is the x-coordinate of the point A on the curve.

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. Section 10.3.2.2.4 has the details of this mapping.

   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 functione.

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

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

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