# 1 Current Hash-Based DRBGs

The current hash-based DRBGs are given in a more traditional algorithmic format. The algorithm description here is confined to the DRBG-boundary. To illustrate the separation expected to be given by the DRBG-boundary a Conceptual API that is accessible external to the cryptographic boundary is given. This should result in a clear deterministic algorithmic description of the DRBG, and guidance on the implementing a secure DRBG-boundary. In addition, it gives a deterministic presentation to the algorithm that are suitable to their deterministic nature and known answer tests.

Before giving the presentation we repeat a necessary table for enforcing security parameters

| Hash Function | Security Strength | Required Minimum Entropy | Entropy Input Lengths | Seed Length for Hash_DRBG |
|---|---|---|---|---|
| SHA-1 | 80 | 128 | $128 - 2^{35}$ | 160 |
| | 112 | 128 | $128 - 2^{35}$ | 176 |
| | 128 | 128 | $128 - 2^{35}$ | 192 |
| SHA-224 | 80, 112, 128 | 128 | $128 - 2^{35}$ | 224 |
| | 192 | 192 | $192 - 2^{35}$ | 256 |
| SHA-256 | 80, 112, 128 | 128 | $128 - 2^{35}$ | 256 |
| | 192 | 192 | $192 - 2^{35}$ | 256 |
| | 256 | 256 | $256 - 2^{35}$ | 320 |
| SHA-384 | 80, 112, 128 | 128 | $128 - 2^{35}$ | 384 |
| | 192 | 192 | $192 - 2^{35}$ | 384 |
| | 256 | 256 | $256 - 2^{35}$ | 384 |
| SHA-512 | 80, 112, 128 | 128 | $128 - 2^{35}$ | 512 |
| | 192 | 192 | $192 - 2^{35}$ | 512 |
| | 256 | 256 | $256 - 2^{35}$ | 512 |

EBB: This is an old table; in particular, some of the seed lengths are wrong. I believe that the current formula is *seedlen* = **max** (*highest_strength* + 64, *outlen*), which means that for SHA-1, the *seedlen* = 192; for SHA-224, *seedlen* = 256; and for SHA-256, *seedlen* = 320. If these are wrong, please correct me.

## Hash_df()

**INPUT:** An *input* buffer and a requested *length*
**OUTPUT:** A output buffer of length *length*; or an **ERROR**

1) If *length* > *max_output_length* return **ERROR**
2) Set *temp* ← *NULL.*
3) Set *n* = *ceil( length/outlen)*
4) For *i=1* to *n.* (*i* an 8-bit counter)

a. Set $temp \leftarrow temp||Hash(i||length||input)$

5) Set *output* $\leftarrow$ leftmost *length-bits* of *temp.*

EBB: This doesn't look that different from the way this is presented in Part 3. Is there some particular part of this specification that needs to be changed from the workshop version of the document? Is there any reason to use "←" instead of "=". Is there any reason that "Set" and "Compute" (see below) are used? A possible revision might be the folowing:

**Input:**

1) *input_string*: The string from which the output will be derived.
2) *number_of_bits_to_return*: The length of the bit string to be returned. The maximum length is implementation depended, but **shall** be ≤ *max_ouput_length*, where *max_output_length* = (255 * *outlen*) bits.

**Output:**

1) *output_string*: A string of the requested length.

**Process:**

1) If *number_of_bits_to_return* > *max_output_length*, return ERROR.
2) Set *temp* = *Null.*
3) $len = \left\lceil \frac{number\_of\_bits\_to\_return}{outlen} \right\rceil$.
4) For *i* = 1 to *len* Comment: *i* is an 8-bit counter.
   Set *temp* = *temp* || **Hash** (*i* || *number of bits to return* || *input_string*).
5) *output_string* = leftmost *number_of_bits_to_return* of temp.
6) Return *output_string.*

Comment [ebb1]: Page: 2
Is this really needed, since the calling procedures control this ?

EBB: In the following procedures, the internal state must not be passed to and from the consuming application, but must be retained within the DRBG boundary. Also, the *entropy_bits* used to derive the seed must not be provided to the procedures by the consuming application. For the instantiation procedure, a *state_handle* is returned and used to the reseed and generation procedure calls. I've shown a *status* code being returned for a successful process, but should this be shown here? The procedures have been specified in accordance with the workshop draft of X9.82. Note that the *mode* parameter (for testing) has been omitted as suggested at the workshop.

## Instantiation Function

**INPUT:** A requested strength ***rstrength***, an entropy input ***seed*** with ***estrength***-bits of entropy and an optional ***input*** bit array, an optional ***pflag*** to indicate prediction resistance

**OUTPUT:** A composite state ***S = {V, C, ctr, seedlen, strength, flag}***, or an **ERROR.**

1) Set ***strength*** to the nearest strength greater than or equal to ***rstrength*** supported by the underlying hash function, if possible; otherwise, return **ERROR.**
2) If ***pflag*** is TRUE and prediction resistance is supported set ***flag*** = ***pflag***; if prediction resistence is not supported return **ERROR**
3) Verify ***strength, estrength*** and ***seedlen*** are appropriate to the underlying hash function as defined by above table, other return **ERROR.**

4) Set *min_entropy* ← *max(128, strength)*
5) If *min_entropy* > *estrength* return **ERROR**
6) Set *min_length* ← *max(outlen, strength)*
7) If *min_length* > *len(seed)* return **ERROR**
8) Set *seed_material* ← *seed* || *input.*
9) Set *seedlen* ← *max(strength + 64, outlen)*
10) Compute *V* ← *Hash_df(seed_material, seedlen)*
11) Compute *C* ← *Hash(0x00||V)*
12) Set *ctr* ← *1*
13) Return *S* ← *{V, C, ctr, seedlen, strength, flag}.*
14)

EBB: Using the above as a model (sort of), I would tend to rewrite the instantiation specification as follows:

**Input from a consuming application:**

1) *requested_strength*: A requested strength for the instantiation.
2) *prediction_resistance_request_flag*: An indication as to whether or not prediction resistance is required for this instantiation . This parameter may be omitted if prediction resistance will never be supported; in this case, step 2 below may be omitted, and the internal state will not contain a *prediction_resistance _flag*. In the following steps, a *prediction_resistance_request_flag* of TRUE indicates that prediction resistance is requested for the instantiation.
3) *personalization_string*: An optional input that provides personalization information. The maximum length of the personalization string is implementation dependent, but **shall** be $\leq 2^{35}$ bits. If a personalization string will never be used, then the input parameter may be omitted, and step 5 may be modified to remove the personalization string.

**Other input:**

1) *entropy_input*: Input containing *min_entropy* bits of entropy. The maximum length of the *entropy_input* is implementation dependent, but **shall** be $\leq 2^{35}$ bits.

**Output to a consuming application:**

1) *status*: The status returned from the procedure. The *status* will indicate **SUCCESS** or an **ERROR**.
2) *state_handle*: A pointer or index that indicates the newly instantiated internal state for subsequent processing using this instantiation.

**Other output/information retained within the DRBG boundary:**

An internal state containing:

1) *V*: An initial value that will be updated for each request for pseudorandom bits.
2) *C*: A constant for the seed period.
3) *reseed_counter*: A counter of the number of requests for pseudorandom bits during the seed period.
4) *strength*: The security strength for the instantiation.
5) *prediction_resistance_flag*: Indicates whether or not prediction resistance requests may be made during the instantiation.

**Process:**

1) Set *strength* to the nearest strength greater than or equal to *requested_strength* that is supported by the underlying hash function, if possible; otherwise, if the *requested_strength* is too large, return **ERROR**.
2) If prediction resistance is supported, set *prediction_resistance_flag* = *prediction_resistance_request_flag*; if prediction resistance is not supported and *prediction_resistance_request_flag* = TRUE, return **ERROR.**
3) Set *min_entropy* = **max** (128, *strength*).
4) Obtain *entropy_input* with at least *min_entropy* bits of entropy. If there is a failure in the *entropy_input* source, return **ERROR.**

Comment: Steps 5-7 contain the instantiation algorithm.

5) Set *seed_material* = *entropy_input* || *personalization_string*.
6) Compute *V* = **Hash_df** (*seed_material*, *seedlen*).
7) Compute *C* = **Hash** (0x00 || *V*).

Comment [ebb2]: Page: 4
Note that since *seedlen* is fixed for a given hash function, this value can be hardcoded into the procedure .

8) Set *reseed_counter* = 1.
9) Get a *state_handle* that will be used to locate the internal state for this instantiation. If an unused internal state cannot be found, return **ERROR**.
10) Set the internal state indicated by *state_handle* to the initial values: *V*, *C*, *reseed_counter*, *strength*, *prediction_resistance_flag*.
11) Return **SUCCESS** and *state_handle*.

EBB: Is this at a high enough level? Note that the mode parameter has been removed, and that whatever handles obtaining the entropy_input in step 4 will check that the entropy source has not failed.

## Reseed Function

**INPUT:** A state $S = \{V, C, ctr, seedlen, strength, flag\}$, an entropy input *seed* with *estrength*-bits of entropy and optional *input* bit array.
**OUTPUT:** A composite state $S = \{V, C, ctr, seedlen, strength, flag\}$ or an **ERROR**

1) Set *min_entropy = max(128, strength)*
2) If *min_entropy > estrength* return **ERROR**
3) Set *min_length = max(128, outlen)*
4) If *min_length > len(seed)* return **ERROR**
5) Set *seed_material* $\leftarrow$ *0x01||V||seed||input.*
6) Compute *V* $\leftarrow$ *Hash_df(seed_material, seedlen).*
7) Compute *C* $\leftarrow$ *Hash(0x00||V)*
8) Return *S* $\leftarrow$ $\{V, C, ctr, seedlen, strength, flag\}$.

EBB: Using the above as a model, I would tend to rewrite the reseed specification as follows:

**Input from a consuming application:**

1) *state_handle*: A pointer or index that indicates the internal state to be reseeded.
2) *additional_input*: An optional input. The maximum length of *the additional_input* is implementation dependent, but **shall** be $\leq 2^{35}$ bits. If *additional_input* will never be used, then the input parameter may be omitted, and step 4 may be modified to remove the

*additional_input*.

**Other input:**

1) *entropy_input*: Input containing *min_entropy* bits of entropy. The maximum length of the *entropy_input* is implementation dependent, but **shall** be ≤ $2^{35}$ bits.
2) Internal state values:
   a) *V*: The latest value of *V*.
   b) *strength*: The security strength for the instantiation.

**Output to a consuming application:**

1) *status*: The status returned from the procedure. The *status* will indicate **SUCCESS** or an **ERROR**.

**Other output/information retained within the DRBG boundary:**

Replaced internal state values:
1) *V*: A new initial value of *V* for the new seed period.
2) *C*: A new constant for the new seed period.
3) *reseed_counter*: A counter of the number of requests for pseudorandom bits during the seed period.

**Process:**

1) Using *state_handle*, obtain the current value of *V* and the *strength* for the instantiation. If *state_handle* indicates an invalid or unused internal state, return **ERROR**.
2) Set *min_entropy* = **max** (128, *strength*).
3) Obtain *entropy_input* with at least *min_entropy* bits of entropy. If there is a failure in the *entropy_input* source, return **ERROR**.

Comment: Steps 4-6 contain the reseed algorithm.

4) Set *seed_material* = 0x01 || *V* || *entropy_input* || *additional_input*.
5) *V* = **Hash_df** (*seed_material*, *seedlen*).
6) Compute *C* = **Hash** (0x00 || *V*).
7) *reseed_counter* = 1.
8) Replace the values of *V*, *C* and *reseed_counter* in the internal state indicated by *state_handle* with the new values.
9) Return **SUCCESS**.

## Generation Function

Note the generation function as given does not allow for a nice implementation of adding new entropy in the case of a counter interval being reached.

**INPUT:** State *S = {V, C, ctr, seedlen, strength, flag}*, a requested *length*, and *rstrength* a requested strength, and optional arguments(*input* array, an optional prediction resistance *pflag*, if *pflag* is TRUE an entropy input *seed* with *estrength*-bits of entropy.

**OUTPUT:** A new state *S = {V, C, ctr, seedlen, strength, flag}* and *output* bit string of length *length*, or an **ERROR.**

1) If *ctr* > *max_ctr_value* return **ERROR**
2) If *rstrength* > *strength* return **ERROR**
3) If *pflag* = TRUE and *flag* = FALSE return **ERROR**
4) If *pflag* = TRUE
    a. Set *min_entropy* ← *max(128, strength)*
    b. If *min_entropy* > *estrength* return **ERROR**
    c. Set *min_length* ← *max(outlen, strength)*
    d. If *min_length* > *len(seed)* return **ERROR**
    e. Set *input* ← *seed||input*
5) If *input* then
    a. Set *w* ← *Hash(0x02||V||input)*
    b. Set $V \leftarrow w + V \bmod 2^{seedlen}$
6) Set *output* ← *HashGen(length, V)*
7) Compute *H* ← *Hash(0x03||V)*
8) Compute $V \leftarrow V + C + H + ctr \bmod 2^{seedlen}$
9) Set *ctr* ← *ctr + 1*
10) Return state *S* ← *{V, C, ctr, seedlen, strength}* and *output.*

## HashGen

**INPUT:** An *input* buffer and a requested *length.*
**OUTPUT:** An *output* buffer of length *length.*

1) Set *m* ← *ceil(length/outlen)*
2) Set *data* ← *V,* and *W* = *NULL*
3) For i = 1 to m
    a. Set *w* ← *Hash(data)*
    b. Set *W* ← *W||w.*
    c. Set *data* ← *data + 1*
4) Set *output* ← leftmost *length* bits of *W*
5) Return *output*

EBB: Using the above as a model. I would tend to rewrite the reseed specification as follows:

**Input from a consuming application:**
1) *state_handle*: A pointer or index that indicates the internal state to be used.
2) *requested_number_of_bits*: The number of pseudorandom bits to be returned from the generation procedure. The maximum number of bits that may be requested is implementation dependent. but **shall** be $\leq 2^{35}$ bits.
3) *requested_strength*: The security strength to be associated with the requested pseudorandom bits.
4) *additional_input*: An optional input. The maximum length of *the additional_input* is implementation dependent. but **shall** be $\leq 2^{35}$ bits. If *additional_input* will never be used. then the input parameter may be omitted. and steps 3b and 4a may be modified to remove the *additional_input*.
5) *prediction_resistance_request_flag*: If TRUE. prediction resistance is requested for the pseudorandom bits to be provided. This parameter may be omitted if prediction resistance

will never be supported; in this case, steps 2 and 3 may be modified to omit the *prediction_resistance_request_flag* and *prediction_resistance_flag*, as appropriate.

**Other input:**

1) Internal state values:
   a) *V*: The latest value of *V*.
   b) *C*: The constant for the seed period.
   c) *reseed_counter*: The number of requests for pseudorandom bits for the current seed period.

**Output to a consuming application:**

1) *status*: The status returned from the procedure. The *status* will indicate **SUCCESS** or an **ERROR**.
2) *pseudorandom_bits*: The pseudorandom bits that were requested.

**Other output/information retained within the DRBG boundary:**

Replaced internal state values:
   1) *V*: The updated value of *V*.
   2) *reseed_counter*: The updated counter for the seed period.

**Process:**

1) Using *state_handle*, obtain the current values of *V*, *C*, *reseed_counter* and *prediction_resistance_flag* for the instantiation. If *state_handle* indicates an invalid or unused internal state, return **ERROR**.
2) Verify that the *requested_number_of_bits* is not too large, the *requested_strength* is ≤ *strength*, and that if *prediction_resistance_request_flag* is TRUE, then *prediction_resistance_flag* is also TRUE. If any of these checks fail, return ERROR.
3) If the *reseed_counter* = the maximum number of requests for the seed period, or the *prediction_resistance_request_flag* is TRUE, then
   a) If a source for *entropy_input* is not available, return an indication that a reseed cannot be performed.
   b) Using *state_handle* and *additional_input* (if provided), reseed the instantiation.
   c) Using *state_handle*, obtain the new values of *V*, *C*, and *reseed_counter*.

   Comment: Steps 4-8 plus **Hashgen** contain the generation algorithm.

4) If *additional_input* has been provided, then
   a) Compute $w$ = **Hash** (0x02 || $V$ || *additional_input*).
   b) Set $V = (V + w) \bmod 2^{seedlen}$.
5) Compute *pseudorandom_bits* = **Hashgen** (*requested_number_of_bits_of_bits*, $V$).
6) Compute $H$ = **Hash** (0x03 || $V$).
7) Set $V = (V + C + reseed\_counter) \bmod 2^{seedlen}$.
8) Set *reseed_counter* = *reseed_counter* + 1

9) Replace the values of *V* and *reseed_counter* in the internal state indicated by *state_handle*.
10) Return **SUCCESS** and *pseudorandom_bits*.

**Hashgen**:

**Input:**

1) *requested_number_of_bits*: The number of pseudorandom bits to be returned from the **Hashgen** routine.
2) *V*: The current value of *V*.

**Output:**

1) *pseudorandom_bits*: The requested pseudorandom bits.

**Process:**

1) Set $m = \left\lceil \frac{requested\_number\_of\_bits}{outlen} \right\rceil$.
2) Set *data* = *V*, and *W* = *NULL*.
3) For *i* = 1 to *m*
   a) Set *w* = **Hash** (*data*).
   b) Set *W* = *W* || *w*.
   c) Set *data* = *data* + 1.
6) Set *pseudorandom_bits* = leftmost *requested_number_of_bits* of *W*
7) Return *pseudorandom_bits*.

## How this works with the Conceptual API

We can express our hash-based DRBG in terms of a conceptual API where we have the notions of exported functions (or public functions) and internal functions (or private functions). To make this more obvious we will adopt a naming convention that external functions will be begin with the HashDRBGXxxx naming convention.

**External/Public Functions**

(Handle, Status) = **HashDRBGInit**(
 integer *requested_strength*,
 [bit array *input*,
 bool *prediction_flag*,
 integer *mode*])

Status = **HashDRBGReseed**(
 Handle *hDRBG*,
 [bit array *input*,
 integer *mode*])

(bit array, Status) = **HashDRBGGenerate**(
 Handle *hDRBG*,
 integer *length*,
 integer *requested_strength*,
 [bit array *input*,
 bool *prediction_flag*])

**Internal/Private Functions**

```
//      This function allocates a state and the associated handle
(Handle, State) = AllocDRBG( )
//      This internal function obtains the state from the handle
(State, Status) = ObtainState(
        Handle hDRBG)
//      This function acquires entropy_input from an RBG.
(bit array, Status) = GetEntropy(
        integer min_length,
        integer max_length)
//      performs the Hash_df algorithm above
bit array = Hash_df(
        bit array input,
        integer length)
//      This function implements the instantiation algorithm above
(State, Status) = hashInstantiate(
        State emptyState
        integer requested_strength,
        bit array seed,
        integer estrength,
        [bit array input,
        bool pflag])
//      This function implements the reseed algorithm above
(State, Status) = hashReseed(
        State currentState,
        bit array seed,
        integer estrength,
        [bit array input])
//      This function implements the generation algorithm above
(State, bit array, status) = hashGenerate(
        State currentState,
        Integer length,
        Integer requested_strength,
        [bit array input,
        Bool prediction_flag,
        Bit array seed,
        Integer estrength)
//      This function implements the hash generation algorithm defined above
bit array = hashGen(
        bit array V,
        Integer length)
```

(Handle, Status) = **HashDRBGInit**(integer *requested_strength*, [bit array *input = NULL*, bool *prediction_flag = 0*, integer *mode*])

1) Set *strength* to the appropriate value based on *requested_strength* defined by above table, if possible otherwise return (NULL, **ERROR**)
2) Set *min_entropy*$\leftarrow$*max(128, strength)*

3) Obtain a new state and the associated handle (hDRBG, emptyState) = **allocDRBG**()
4) Obtain the appropriate entropy *(entropy_input, Status) =* **GetEntropy**(*min_entropy,max_entropy_size)*. If *Status* is an error condition return (NULL, **ERROR**)
5) Perform the instantiation function *(newState, Status)* = **hashInstantiate**(*emptyState, strength, entropy_input, min_entropy, input, prediction_flag*). If *Status* is an error condition return (NULL, **ERROR**)
6) Return (*hDRBG, STATUS_OK*)

Status = **HashDRBGReseed**(Handle *hDRBG*, [bit array *input=NULL*, integer *mode*])

1) Obtain the underlying state (*currentState, Status*) = **ObtainState**(*hDRBG*). If *Status* is an error condition return **ERROR**.
2) Set *min_entropy = max(128, currentState.strength)*
3) Obtain the appropriate entropy (*entropy_input, Status) =* **GetEntropy**(*min_entropy,max_entropy_size)*. If *Status* is an error condition return **ERROR.**
4) Call the internal reseed function *(newState, Status)* = **hashReseed***(currentState, entropy_input, min_entropy, input)*.
5) Return *STATUS_OKs*

(bit array, Status) = **HashDRBGGenerate**(Handle *hDRBG*, integer *length*, integer *requested_strength,* [bit array *input = NULL*, bool *prediction_flag= FALSE*])

1) Obtain the underlying state (*currentState, Status*) = **ObtainState**(*hDRBG*). If *Status* is an error condition return (NULL, **ERROR**).
2) If *currentState.ctr > max_counter* return (NULL, **ERROR**).
3) If *(prediction_flag = TRUE)* then
   a. If *currentState.flag* = FALSE) return (NULL, **ERROR)**
   b. Else Set *min_entropy = max(128, currentState.strength)*
   c. *(entropy_input, Status)* = **GetEntropy**(*min_entropy, max_entropy_size)*. If *Status* is an error condition return (NULL, **ERROR**).
   d. Set e*strength = min_entropy*
4) Else *entropy_input* = NULL, *estrength = 0*
5) Call the internal generation (*newState, output, status)* = **hashGenerate**(*currentState, length, requested_strength, input, prediction_flag, entropy_input, min_entropy)*. If *status* represents an error condition return (NULL, **ERROR**).
6) If *(newState.ctr > max_counter)* set *(newState, status)* = **HashDRBGReseed**(*newState,* NULL, *mode)*.
7) Return *(output, status)*.