# ANNEX D: (Informative) DRBG Selection

[This will need to be revised, based on the DRBGs that are retained and the content of Part 4.]

## D.1 Choosing a DRBG Algorithm

Almost no application or system designer starts with the primary purpose of generating good random bits. Instead, he typically starts with some goal that he wishes to accomplish, then decides on some cryptographic mechanisms, such as digital signatures or block ciphers that can help him achieve that goal. Typically, as he begins to understand the requirements of those cryptographic mechanisms, he learns that he will also have to generate some random bits, and that this must be done with great care, or he may inadvertently weaken the cryptographic mechanisms that he has chosen to implement. At this point, there are two things that may guide the designer's choice of a DRBG:

a. He may already have decided to include a set of cryptographic primitives as part of his implementation. By choosing a DRBG based on one of these primitives, he can minimize the cost of adding that DRBG. In hardware, this translates to lower gate count, less power consumption, and less hardware that must be protected against probing and power analysis. In software, this translates to fewer lines of code to write, test, and validate.

   For example, a module that generates RSA signatures has available some kind of hashing engine, so a hash-based DRBG is a natural choice.

b. He may already have decided to trust a block cipher, hash function, keyed hash function, etc., to have certain properties. By choosing a DRBG based on similar properties, he can minimize the number of algorithms he has to trust.

   For example, an AES-based DRBG might be a good choice when a module provides encryption with AES. Since the DRBG is based for its security on the strength of AES, the module's security is not made dependent on any additional cryptographic primitives or assumptions.

c. Multiple cryptographic primitives may be available within the system or application, but there may be restrictions that need to be addressed (e.g.,code size or performance requirements).

The DRBGs specified in this Standard have different performance characteristics, implementation issues, and security assumptions.

## D.2 DRBGs Based on Hash Functions

Two DRBGs are based on any Approved hash function: **Hash_DRBG**, and **HMAC_DRBG**. A hash function is composed of an initial value, a padding mechanism and a compression function; the compression function itself may be expressed as **Compress** $(I, X)$, where $I$ is the initial value, and $X$ is the compression function input. All of the cryptographic security of the hash function depends on the compression function.

and the compression is by far the most time-consuming operation within the hash function.

The hash-based DRBGs in this Standard allow for some tradeoffs between performance, security assumptions required for the security of the DRBGs, and ease of implementation.

### D.2.1  Hash_DRBG


### D.2.2   HMAC_DRBG

**HMAC_DRBG** is a DRBG whose security is based on the assumption that HMAC is a pseudorandom function.  [I think the following needs to be either augmented to complete the ideas, or removed.] The security of **HMAC_DRBG** is based on an attacker getting sequences of no more than to $2^{35}$ bits, generated by the following steps:

*temp* = the *Null* string.

While (**len** (*temp*) < *requested_no_of_bits*:

V = **HMAC** (*K*, *V*).

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

The steps in the "While" statement iterate $\lceil$ *requested_no_of_bits/outlen* $\rceil$ times. Intuitively, so long as *V* does not repeat, any algorithm that can distinguish this output sequence from an ideal random sequence can be used in a straightforward way to distinguish HMAC from a pseudorandom function.

Between these output sequences, both *V* and *K* are updated using the following steps (assuming no additional inputs):

*K* = **HMAC** (*K*, (*V* || 0x01)) = **Hash** (opad (*K*) || **Hash** (ipad (*K*) || (*V* || 0x01))).

*V* = **HMAC** (*K*, *V*) = **Hash** (opad (*K*) || (**Hash** (ipad (*K*) || *V*)).

where:

*K* and *V* are *outlen* bits long.

**opad** (*K*) is *K* exclusive-ored with (*inlen*/8) bytes of 0x5c, for a total of *inlen* bits,

**ipad** (*K*) is *K* exclusive-ored with (*inlen*/8) bytes of 0x36, for a total of *inlen* bits,

*outlen* is the length of the hash function output block, and

*inlen* is the length of the hash function input block.

### D.2.2.1   Implementation Properties

The only thing required to implement this DRBG is access to a hashing engine. However, the performance of the implementation will improve enormously (by about a factor of two!) with either a dedicated **HMAC** engine, or direct access to the hash function's underlying compression function.  The "critical state values" on which **HMAC_DRBG** depends for its security (*K* and *V*) take up 2\**outlen* bits in the most

compact form, but for reasonable performance, 3*$outlen$ bits are required in order to precompute padded values.

### D.2.2.2　Performance Properties

Each *outlen*-bit piece of the requested pseudorandom output requires two compression function calls to perform the **HMAC** computation. Each output request also incurs another six compression function calls to update the state.

Note that an implementation that has access only to a high-level hashing engine loses another factor of two in performance; if the performance of the DRBG is important, **HMAC_DRBG** requires either a dedicated **HMAC** engine or access to the compression function that underlies the hash function. However, if performance is not an important issue, the DRBG can be implemented using nothing but a high-level hashing engine.

## D.2.3 Summary and Comparison of Hash-Based DRBGs

### D.2.3.1 Security

It is interesting to contrast the two ways that the hash function is used in these DRBGs.

**HMAC_DRBG:**

$V_1 = $ **HMAC** $(K, V_0) = $ **Hash** (opad $(K) \parallel$ (**Hash** (ipad $(K) \parallel V_0$))).

$V_2 = $ **HMAC** $(K, V_1) = $ **Hash** (opad $(K) \parallel$ (**Hash** (ipad $(K) \parallel V_1$))).

$V_3 = $ **HMAC** $(K, V_2) = $ **Hash** (opad $(K) \parallel$ (**Hash** (ipad $(K) \parallel V_2$))).

etc

as specified in Annex E.2.2.

The adversary knows many specific bits of the input to the final compression function whose output he sees; for SHA-256, the compression function takes a total of 768 bits of input, and the adversary knows 256 of those bits[1]. (This is worse for SHA-1 and SHA-384.) On the other hand, the adversary doesn't even know the exclusive-or relationships for *outlen* bits of the message input. In the case of SHA-256, this means that 256 bits are unknown.

**HMAC_DRBG** allows an adversary to precisely know many bits of the input to the compression functions, but not to know complete exclusive-or or additive relationships between these bits of input.

---

[1] The innermost hash function provides *outlen* bits of input after its two compression function calls on **ipad** $(K)$ and $V$. The outermost hash function also requires two compression functions: the first operates on **opad** $(K)$ and produces *outlen* bits that are used as the chaining value for the final compression function on the result from the innermost hash function concatenated with the hash function padding. Therefore, the input to the final compression function is the length of the chaining value (*outlen* bits) + the length of the ouput from the innermost hash function (*outlen* bits) + the length of the padding (*inlen - outlen* bits). In the case of SHA-256, where *inlen* = 512, and *outlen* = 256, the length of the input to the last compression function is 768 bits, of which only the padding bits are known (256 bits).

### D.2.3.2 Performance / Implementation Tradeoffs

The following performance and implementation tradeoffs should be considered when selecting a hash-based DRBG with regard to the overhead associated with requesting pseudorandom bits, the cost of actually generating *outlen* bits (not including the overhead), and the memory required for the critical state values for each DRBG. The overhead is, essentially, the cost of updating the state prior to the next request for pseudorandom bits. The cost of generating each *outlen* block of bits of output should be multiplied by the number of *outlen*-bit blocks of output required in order to obtain the true cost of pseudorandom bit generation. Both the overhead and generation costs assume that prediction resistance and reseeding are not required, and that additional input is not provided for the request; if this is not the case, the costs are increased accordingly. Note that the memory requirements do not take into account other information in the state that is required for a given DRBG.

__HMAC_DRBG__ (with access to the hash function's compression function):

Request overhead: six compression functions[2].

Cost for *outlen* bits of pseudorandom output: two compression functions.

Memory required for the critical state values $K$ and $V$: 3*$outlen$ bits when precomputation is used .

__HMAC_DRBG__ (hash engine access only):

Request overhead: eight compression function calls[3].

Cost for *outlen* bits of pseudorandom output: four compression functions[4].

Memory required for the critical state values $K$ and $V$: 2*$outlen$ bits, since precomputation is unavailable.

Additional inputs provided during pseudorandom bit generation add considerably to the request overhead. Instantiation and reseeding are somewhat more expensive than pseudorandom output generation; however, these relatively rare operations can afford to be somewhat more expensive to minimize the chances of a successful attack.


## D.3 DRBGs Based on Block Ciphers

### D.3.1 The Two Constructions: CTR and OFB

This standard describes DRBGs based on block ciphers using the CTR-mode. The CTR mode guarantees that short cycles cannot occur in a single output request. The security of

---

[2] Two compression functions for each HMAC computation, and two compression functions for precomputation.

[3] There are two HMAC computations, each requiring two hash function calls. Each hash computation requires two compression function calls.

[4] The single HMAC computation requires four compression functions as explained in the previous footnote.

the DRBGs relates in a very simple and clean way to the security of the block cipher in its intended applications. This is a fundamental difference between the CTR_DRBG and a hash function-based DRBG, where the DRBG's security is ultimately based on pseudorandomness properties that do not form a normal part of the requirements for hash functions. An attack on any of the hash-based DRBGs does not necessarily represent a weakness in the hash function; however, for these block cipher-based constructions, a weakness in the DRBG is directly related to a weakness in the block cipher.

## D.3.2 Choosing a Block Cipher

The choice of the block cipher algorithm to be used is a security issue. At present, only TDEA and AES are approved block cipher algorithms.

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

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

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

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

2. For three-key TDEA with 168-bit keys and 64-bit blocks, things are a bit different: There are $2^{16}$ **Generate (...)** requests allowed, and a maximum of $2^{13}$ blocks (i.e., $2^{16}$ bytes = $2^{19}$ bits) generated per **Generate (...)** request. (Note that this breaks the more general model in this document of assuming $2^{security\_level/2}$ innocent operations.) In this case:

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

   b. The expected probability of an internal collision in a sequence of $2^{13}$ 64-bit blocks is about $2^{-38}$. Thus, the probability of ever seeing an internal collision in $2^{16}$ output sequences is still an acceptably low $2^{-22}$. (Note that if more **Generate (...)** requests are allowed, there would be an unacceptably high

probability of this happening, leading to an efficient distinguisher between this DRBG's outputs and ideal random outputs.

c. The probability of a key colliding between any two of the $2^{16}$ **Generate (...)** requests is about $2^{-136}$, which is negligible.

To summarize: block size matters. The limits on the numbers of **Generate (...)** requests and the number of output bits per request require frequent reseeding of the DRBG. Furthermore, the limits guarantee that even with reseeding, an adversary that is given a really long sequence of DRBG outputs from several reseedings cannot distinguish that output sequence from random reliably. The CTR_DRBG used with TDEA is suitable for low-throughput applications, but not for applications requiring really large numbers of DRBG outputs. **For concreteness, if an application is going to require more than $2^{32}$ output bytes ($2^{35}$ bits) in its lifetime, that application should not use a block cipher DRBG with TDEA.**

### D.3.3   Conditioned Entropy Sources and the Derivation Function

[Some or all of this section probably belongs in Part 4]

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