E.1 Choosing a DRBG Algorithm

Almost no system designer starts with the idea that he's going to generate 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 block cipher, hash function, keyed hash function, etc., as part of his implementation. By choosing a DRBG based on one of these mechanisms, 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 designer of a module that does RSA signatures probably already has available some kind of hashing engine, so one of the three hash-based DRBGs 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 of these mechanisms, he can minimize the number of algorithms he has to trust.

   For example, a designer of a module that provides encryption with AES can implement an AES-based DRBG. 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.

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

E.2 DRBGs Based on Hash Functions

Three DRBGs are based on a Approved hash function: **Hash_DRBG**, **KHF_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 three 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.

E.2.1 Hash_DRBG

Hash_DRBG is closely related to the DRBG specified in FIPS-186-2, and can be seen as an updated version of that DRBG that can be used as a general-purpose DRBG. Although a formal analysis of this DRBG is not available, it is clear that the security of the DRBG depends on the security of **Hashgen**. Specifically, an attacker can get a large number of values:

Hash (*V*), **Hash** (*V*+1), **Hash** (*V*+2), ...

If the attacker can distinguish any of these sequences from a random sequence of values, then the DRBG can be broken.

E.2.1.1 Implementation Issues

This DRBG requires a hash function, some surrounding logic, and the ability to add numbers modulo $2^{seedlen}$, where *seedlen* is the length of the seed. **Hash_DRBG** also uses **hash_df** internally when instantiating, reseeding, or processing additional input. Note that **hash_df** requires only access to a general-purpose hashing engine and the use of a 32-bit counter. The "critical state values" on which the **Hash_DRBG** depends for its security (*V*, *C* and *reseed_counter*) require *seedlen* + *outlen* + 32 bits of memory[1].

E.2.1.2 Performance Properties

Each time that **Hash_DRBG** is called, a compression function computation is required for each *outlen* bits of requested output (or portion thereof), where *outlen* is the size of the hash function output block. For example, if *outlen* = 160, and 360 bits of pseudorandom data are requested, three compression function calls are made (two to produce the first 320 bits, and a third from which to select the remaining 40 bits. In addition, there is a certain amount of overhead to updating the state in order to achieve backtracking resistance; this requires one compression function call and some additions modulo $2^{seedlenn}$, plus the update of *reseed_counter*. For the above example, a total of four compression function calls are required, three to generate the requested output bits, and one to update the state.

E.2.2 HMAC_DRBG

HMAC_DRBG is a DRBG whose security is based on the assumption that HMAC is a pseudorandom function. The security of **HMAC_DRBG** is based on an attacker getting sequences of up to 2^{35} bits, generated by the following steps:

temp = the *Nu*ll 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

---

[1] *V* is *seedlen* bits long, *C* is *outlen* bits long (where *outlen* is the length of the hash function output block), and *reseed_counter* is a maximum of 32 bits in length.

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.

E.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 values.

E.2.2.2 Performance Properties

HMAC_DRBG is about a factor of two slower than the other two hash-based DRBGs for long bitstrings produced by a single request. That is, 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.

E.2.3 KHF_DRBG

KHF_DRBG is also based on a hash-function-based pseudorandom function that requires only one compression function call per *outlen* bits of requested DRBG output. This DRBG has essentially the same structure as **HMAC_DRBG**, but with the substitution of the **KHF** function in place of the **HMAC** function. The DRBG uses two keys, with a total key of *outlen* + *inlen* - 72 bits. Thus, with SHA-256, **KHF_DRBG** has a total key size of 696

bits[2]. The **KHF** function is only defined for *outlen*-bit inputs (the length of V). It is defined as:

$$\mathbf{KHF}\ (K_0, K_1, V) = \mathbf{Hash}\ (\mathbf{pad}\ (K_0) \| (K_1 \oplus\ \mathbf{pad}\ (V))).$$

where,

K_0 is *outlen* bits in length,

K_1 is *inlen*-72 bits in length, and

V is *outlen* bits in length,

pad (K_0) is K_0 padded to *inlen* bits, and

pad (V) is V padded to *inlen*-72 bits.

These specific lengths have been chosen to allow the use of general-purpose hashing engines with a minimal loss in performance. Note that the 72 bits not included in the length of K_1 and **pad** (V) will consist of a byte to mark the end of the value (0x80), plus 8 bytes that are used to indicate the length of the value.

The basic design principle of **KHF** is to put as many unknown bits into the input of the hash function as possible, without impacting performance too heavily.

E.2.3.1 Implementation Issues

All that is needed to implement **KHF_DRBG** is a general-purpose hashing engine. However, an implementation that doesn't have access to the underlying compression function of the hash function will suffer a "factor of two" performance penalty. An implementation also uses **hash_df**, which itself uses only general-purpose hashing calls and a 32-bit counter. The total memory needed to hold the "critical state values" on which **KHF_DRBG** depends for its security (K_0, K_1 and V) is 2**outlen* + *inlen* - 72 bits.

E.2.3.2 Performance Characteristics

A single request for pseudorandom bits always incurs some substantial overhead to update the state after the pseudorandom bits are generated (i.e., $\lceil (outlen+inlen-72)/outlen \rceil + 1$ compression function calls); using SHA-1, the overhead is five compression function calls; using SHA-256, the overhead is four compression function calls. Each *outlen* bits of pseudorandom output within a single request is produced with a single compression function call.

E.4 Summary and Comparison

E.4.1 Security

It is interesting to contrast the three ways that the hash function is used in these three DRBGs:

Hash_DRBG:

Hash (V), **Hash** (V+1), **Hash** (V+2)...

---

[2] *outlen* = 256, *inlen* = 512, and *outlen* + *inlen* - 72 =256 + 512 - 72 = 696 bits.

The only unknown input into the compression function used by the hash function is this sequence of secret values, $V+i$. Since the initial value of the hash function is publicly known, the attacker is given full knowledge of all but *seedlen* bits of input into the compression function, and knowledge of the close relationship between these inputs, as well.

HMAC_DRBG:

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

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

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

etc

as specified in Annex E.2.2.

The attacker 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 attacker knows 256 of those bits[3]. (This is worse for SHA-1 and SHA-384.) On the other hand, the attacker doesn't even know the exclusive-or relationships for *outlen* bits of the message input[4]. In the case of SHA-256, this means that 256 bits are unknown.

KHF_DRBG:

$V_1 =$ **KHF** $(K_0, K_1, V) =$ **Hash** (**pad** $(K_0) \| (K_1 \oplus$ **pad** $(V_0))$.

$V_2 =$ **KHF** $(K_0, K_1, V) =$ **Hash** (**pad** $(K_0) \| (K_1 \oplus$ **pad** $(V_1))$.

$V3 =$ **KHF** $(K_0, K_1, V) =$ **Hash** (**pad** $(K_0) \| (K_1 \oplus$ **pad** $(V_2))$.

etc.

as defined in Annex E.2.3.

The attacker knows only 72 bits of the input to the compression function, but he also knows exactly what the exclusive-or differences are between these inputs. Thus, if there is a differential attack on the compression function using only known (not chosen) *outlen*-bit differences in the input block, which allows a distinguisher on the whole hash function that can be checked with less than 2^{outlen} total work, then the DRBG is broken. Good hash functions seem to be quite hard to attack in this very restricted way.

It is clear that **Hash_DRBG** makes the strongest assumptions on the strength of the compression function. Although they are not precisely comparable, **HMAC_DRBG** seems

---

[3] 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).

[4] K and V are each *outlen* bits in length, and when exclusive-ored will produce an *outlen* bit result.

to make somewhat weaker assumptions on the compression function than **KHF_DRBG**. Specifically, **HMAC_DRBG** allows an attacker 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. **KHF_DRBG** allows an attacker to know only 72 bits of input to the compression function, but to precisely know (but not choose) the complete exclusive-or relationships between these inputs.

E.4.2 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* 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.

Hash_DRBG:

Request overhead: one compression function and several additions mod $2^{seedlen}$.

Cost for *outlen* bits of pseudorandom output: one compression function.

Memory required for the critical state values *V*, *C* and *reseed_counter*: *seedlen* + *outlen* + 32.

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

Request overhead: six compression functions[5].

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[6].

Cost for *outlen* bits of pseudorandom output: four compression functions[7].

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

---

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

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

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

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

Request overhead: five to seven compression functions (depends on the values of *inlen* and *outlen*)[8].

Cost for *outlen* bits of pseudorandom output: one compression function.

Memory required for the critical state values K_0, K_1 and V: *inlen*+2**outlen* bits.

KHF_DRBG (hash engine access only):

Request overhead: eight to twelve compression functions (depends on the values of *inlen* and *outlen*)[9].

Cost for *outlen* bits of pseudorandom output: two compression functions[10].

Memory required for the critical state values K_0, K_1 and V: 2**outlen* + *inlen* - 72 bits.

For all these DRBGs, additional inputs provided during pseudorandom bit generation add considerably to the request overhead. For all three DRBGs, instantiation and reseeding is somewhat more expensive than psueudorandom output generation; however, these relatively rare operations can afford to be somewhat more expensive to minimize the chances of a successful attack.

---

[8] For example, for SHA-256, *inlen* = 512 and *outlen* = 256. The update process requires three calls to the KHF process to generate the *outlen* + *inlen* - 72 = 696 bits. Assuming that K_0 has been precomputed and has not changed, the hash function call in the **KHF** process requires only one compression function call; thus, a total of three compression function calls are required during the three **KHF** function calls. The **hash_df** function call within the update process also requires three hash function calls, each requiring one compression function call (assuming that the input_string is is null or reasonably short). Therefore, in the case of SHA-256, a total of six compression function calls are required.

[9] For example, for SHA-256, *inlen* = 512 and *outlen* = 256. The update process requires three calls to the KHF process to generate the *outlen* + *inlen* - 72 = 696 bits. Each KHF call hashes 2**inlen* - 72 bits, requiring two compression functions for each KHF process. This results in a total of six compression function calls. The **hash_df** function call within the update process also requires three hash function calls, each requiring one compression function call (assuming that the input_string is is null or reasonably short).

[10] As stated in the previous footnote, each call to the KHF process requires two compression funtion calls.