9.12 Choosing a DRBG Algorithm

Almost no system designer starts out with the idea that he's going to generate good random bits. Instead, he typically starts with some goal 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 comes 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 he has chosen to implement. At this point, there are two things that may guide the designer's choice of 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 does 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 seven DRBGs specified in this standard have different performance characteristics, implementation issues, and security assumptions.

9.12.1 DRBGs Based on Hash Functions

Three DRBGs are based on some underlying approved hash function: Hash_DRBG, HMAC_DRBG, and KHF_DRBG. These three DRBGs allow for some tradeoffs between performance, security assumptions required for the security of the DRBGs, and ease of implementation.

9.12.1.1 Hash_DRBG

Hash_DRBG is closely related to the DRBG specified in FIPS-186, and can be seen as an updated version of that DRBG, for use as a general-purpose DRBG. Although we do not have a formal analysis of this DRBG, it is clear that the security of the DRBG depends on

the security of the hash function, and that, more specifically, an attacker can get a large number of values:

hash(V),hash(V+1),hash(V+2),...

If the attacker can distinguish this kind sequence from a random sequence of values, then he can break the DRBG.

9.12.1.1.1 Implementation Issues

In implementation terms, this DRBG requires a hash function and some surrounding logic, and the ability to add numbers modulo 2^{seedlen}, where seedlen is the length of the seed maintained. Hash_DRBG also makes use of hash_df internally, but only 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 DRBG state requires no more that seedlen+64+hlen bits, and can save a few bits in implementations that limit the number of calls to the DRBG.

X.1.1.2 Performance Properties

Each time the Hash_DRBG is called, there is a certain amount of overhead in updating the seed after the generation (this requires one hash computation and some additions modulo 2^{seedlen}). This is required to achieve backtracking resistance. When the DRBG is called to generate a long pseudorandom bitstring, it requires one hash compression call for each hlen bits of output, where hlen is the size of the output of the hash function, and updating a counter. The DRBG is thus quite efficient.

If the Hash_DRBG is called to generate a single hlen-bit output in a call, with no additional input, it requires two hash compression functions.

Note that both of these assume that seedlen <= inlen-80. (Thus, for SHA256, seedlen would have to be less than 432 bits to get this performance.) The Hash_DRBG allows a seedlen up to inlen, which for SHA256 is 512 bits, but this size of seed will cause a factor of two slowdown in the performance of the DRBG, with no known security improvement.

X.1.2 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^{32} bytes, generated by the equation:

```
tmp = ""
for i = 0 to outputBlocks -1:
   X = hmac(K,X)
   tmp = tmp || X
```

Intuitively, so long as X 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 X and K are updated by the formula (assuming no additional inputs)

```
K = hmac(K,X||0x01)
X = hmac(K,X)
```

X.1.2.1 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 DRBG state takes up 2*hlen bits in its most compact form, but for reasonable performance, 3*hlen bits are required.

X.1.2.2 Performance Properties

In performance terms, 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 hlen-bit piece of the output requires two compression function calls to generate. Each output request incurs another five compression function calls.

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

X.1.3 KHF_DRBG

KHF_DRBG is also based on a hash-function-based PRF construction, but one which requires only one compression function call per output. It has essentially the same structure as HMAC_DRBG, but with the substitution of the KHF function. This function is only defined for hlen-bit inputs, and has a total key of hlen+inlen-72 bits. (Thus with SHA256, it has a total key of 696 bits.) It is defined as:

F(K0,K1,X) = Compress(K0,K1 xor pad(X))
where
K0 is hlen bits
K1 is inlen-72 bits

where pad(X) is the hlen-bit X, padded to inlen-72 bits. (This specific length is chosen to allow the use of general-purpose hashing engines with a minimal loss in performance.)

The basic design principle of F() is to put as many unknown bits into the input of the compression function as we can, without impacting performance too heavily.

X.1.3.1 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 will suffer a factor of two performance penalty. (KHF is defined with a hlen-bit key K0', such that hash(K0')==K0.) An implementation also makes use of hash_df(), which itself uses only general-purpose hashing calls and a 32-bit counter. The total memory needed to hold the KHF_DRBG state is hlen*2+inlen bits.

X.1.3.2 Performance Characteristics

A single request always incurs some substantial overhead--using SHA1, the overhead is five compression function calls; using SHA256, the overhead is three compression function calls. Each hlen bits of output within a single request is produced with a single compression function call.

X.1.4 Summary and Comparison

X.1.4.1 Security

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

Hash_DRBG:

Compress(I,V), Compress(I,V+1), Compress(I,V+2)

Here, the only unknown input into Compress() is this sequence of secret values, V. 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.

KHF_DRBG:

X1 = Compress(K0,pad(K1 xor X0)))
X2 = Compress(K0,pad(K1 xor X1)))
X3 = Compress(K0,pad(K1 xor X2)))
etc.
where pad(t) is the hash function's standard message padding scheme, and K1 xor t means that t is zero padded on the right to the same length as K1, and then XORed with K1.

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

HMAC_DRBG:

X1 = Compress(K0,pad(Compress(K1,pad(X0))))
X2 = Compress(K0,pad(Compress(K1,pad(X1))))
X3 = Compress(K0,pad(Compress(K1,pad(X2))))
etc

where pad(t) is the standard padding and length extension of the hash on input t. Here, the attacker knows many specific bits of the input to the compression function whose output he sees--for SHA256, the compression function takes a total of 768 bits of input, and the attacker knows 256 of those bits. (This is worse for SHA1 and SHA384.) On the other hand, the attacker doesn't even know XOR relationships for hlen bits of the message input.

It is clear that Hash_DRBG makes the strongest assumptions on the strength of the compression function, especially when seedlen = hlen,

which is the minimum value allowed. Although it's not precisely comparable, HMAC_DRBG seems 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 function, but not to know complete XOR or additive relationships between these bits of input. KHF_DRBG allows an attacker to precisely know only 72 bits of input to the compression function, but to precisely know (but not choose) the complete XOR relationships between these inputs.

X.1.4.2 Performance / Implementation Tradeoffs

Hash_DRBG (seedlen < inlen-80)

Request Overhead: one compress, several additions mod 2^{seedlen}
Cost for hlen Bits: one compress
State Size: seedlen + hlen + 64

Hash_DRBG (seedlen == inlen)

Request Overhead: two compress, several additions mod 2^{seedlen}
Cost for hlen Bits: two compress
State Size: seedlen + hlen + 64

HMAC_DRBG (compression function access)

Request Overhead: five compress
Cost for hlen Bits: two compress
State Size: hlen*3 bits

HMAC_DRBG (hash engine only)

Request Overhead: four compress
Cost for hlen Bits: three compress
State Size: hlen*2 bits

KHF_DRBG (compression function access)

Request Overhead: six to ten compress (depends on inlen and hlen)
Cost for hlen Bits: two compress
State Size: inlen+2*hlen bits

KHF_DRBG (hash engine only)

Request Overhead: three to five compress (depends on inlen and hlen)
Cost for hlen Bits: one compress

State Size: inlen+2*hlen bits

For all these DRBGs, additional inputs add considerably to the request overhead. For all three DRBGs, instantiation and reseeding is somewhat more expensive than output generation; our assumption here is that these relatively rare operations can afford to be somewhat more expensive to minimize the chances of successful attack.

X.2 DRBGs Based on Block Ciphers

[[This is all assuming my block cipher based schemes are acceptable to the NSA guys doing the review.--JMK]]

X.2.1 The Two Constructions: CTR and OFB

This standard describes two classes of DRBG based on block ciphers: One uses the block cipher in OFB-mode, the other in CTR-mode. There are almost no security differences between these two DRBGs; CTR mode guarantees that short cycles cannot occur in a single output request, while OFB-mode simply guarantees that short cycles will have an extremely low probability. OFB-mode makes slightly less demanding assumptions on the block cipher, but the security of both 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 these DRBGs and the ones based on hash functions, where the DRBG's security was ultimately based on pseudorandomness properties that don't form a normal part of the requirements for hash functions. An attack on any of the hash-based DRBGs would not necessarily represent a weakness in the hash function; for these constructions, a weakness in the DRBG is directly related to a weakness in the block cipher.

To be a little more concrete, each request for pseudorandom bits made without any additional input produces up to 2^{32} bytes under AES, or 2^{16} bytes under TDEA. Each request leads to the generation of these required bits, followed by a rekeying done using some additional output bits.

For CTR mode, suppose there is an attack that allows the attacker to distinguish the outputs from random. This can be used to distinguish the block cipher from random in a chosen plaintext attack with the same text requirements and resources. For OFB mode, this is also true, unless we happen to land in a short cycle. With the limits on output sizes per request imposed for AES and TDEA, this happens with negligible probability. (2^{16} outputs are allowed with TDEA; this leaves approximately a 2^{-48} probability of a short cycle. With

AES, 2^{32} outputs are allowed; this leaves approximately a 2^{-96} probability of a short cycle.)

At the end of each request, the block cipher and IV are regenerated by the DRBG. Suppose the selection of this key led to a bias in the next output. Then, an attacker would again have a straightforward way to convert that attack into one that demonstrated a weakness in CTR or OFB modes, and thus in the underlying block cipher.

Assuming the outputs are indistinguishable from random, the maximum of 2^{64} rekeyings lead to the following rough probabilities of a short cycle:

Cipher	Total State	P(cycle) in 2^{64} Tries
AES128	(256)	2^{-128}
AES192	(384)	2^{-256}
AES256	(512)	2^{-384}
2KTDEA	(176)	2^{-48}
3KTDEA	(232)	2^{-104}

For all of these, the cycling probabilities in 2^{64} requests are negligible.

X.2.1.1 Implementation Issues

The only thing required to implement the raw DRBGs is access to a block cipher, both the key schedule and the encryption function. (Even the decryption function can be ignored here, which is somewhat helpful for ciphers like AES, whose decryption function is a bit different than their encryption function.)

In order to implement the full DRBG, with the ability to be instantiated and reseeded with free-form input strings, rather than only with full-entropy strings, we must also implement the bc_df. This currently amounts to CBC-MAC and OFB-mode, but we may change it soon. [[!!!--JMK]]

X.2.1.1 Performance Characteristics

The block cipher based DRBGs have excellent performance. For a single request, the overhead is between three and four block encryptions and one rekeying. Each blocklen-bit piece of the output (where blocklen is the block size of the block cipher) is generated with a single encryption operation.

These DRBGs can be used for any block cipher with blocklen>= 64 and keylen>=112. (A block cipher such as Skipjack, with an 80 bit key and 64-bit block, would not work here; the probability of a short cycle in 2^{64} requests would be about 2^{-16}, far too high to be acceptable!) However, we note that block ciphers with extremely slow key schedules, such as Blowfish and Khufu, are not very practical with these DRBGs, because the per-request overhead will be very high.

X.3 DRBGs Based on Hard Problems

[[Okay, so here's the limit of my competence. Can Don or Dan or one of the NSA guys with some number theory/algebraic geometry background please look this over? Thanks! --JMK]]

Some DRBGs are based in an intuitive and powerful way on problems believed to be hard, on which much of modern cryptography is based. The DUAL_EC_DRBG and MICALI_SCHNORR_DRBG are based on the difficulty of the elliptic curve discrete log problem, and factoring, respectively. These schemes are several orders of magnitude slower than those based on hash functions and block ciphers, and so aren't appropriate for most systems. However, in some devices, hash functions must be computed by low-powered general-purpose processors, while modular exponentiation or point multiplication is taken care of by special-purpose hardware. In other devices and applications, random numbers are needed at an extremely low rate, e.g., only when a new keypair is being generated, or only once in a great while for a signature or key agreement. In such cases, these DRBGs may be reasonable.

The security of these DRBGs reduces to that of the underlying hard problem. However, both of these DRBGs do rely to some extent on a hash function and the hash_df function, to instantiate them from an unguessable seed to a random starting state, and to process additional input.

[[Do we have a simple proof of this for these two schemes?]]

We note two other important points about the security of these schemes:

a. Side-channel attacks are typically much easier against this kind of algorithm than against symmetric algorithms. Thus, implementations of these DRBGs need to resist timing, power, radiation, and differential fault analysis. (Note that all algorithms are susceptible to these attacks in unprotected hardware; it's just that modular exponentiation and point multiplication are somewhat easier targets for these attacks.)

X.3.1 Dual_EC_DRBG

The DUAL_EC_DRBG relies for its security on the difficulty of the elliptic curve discrete log problem--given (P,xP), determine x. Widely used signature and key agreement schemes are based on this problem, as well. A very conservative system design which had few performance requirements on its random number generation mechanism might thus choose the DUAL_EC_DRBG as its DRBG. This would ensure that the security of the whole application or system relied very cleanly on the difficulty of this one problem.

[[I'm really blowing smoke here. Would someone with some actual understanding of these attacks please save me from diving off a cliff right here? --JMK]]

One important security point with respect to DUAL_EC_DRBG: The best known ways to compute discrete logs involve a massive precomputation, after which it is in general easy to compute discrete logs. Thus, an attacker who does this massive precomputation (equal to the difficulty of violating the security level) can break all instances of this DRBG being used with the same curve. The Hash_DRBG has a similar property for minimum seedlen for each security level. The HMAC_DRBG and KHF_DRBGs, and the block-cipher-based DRBGs, appear to require a much larger precomputation than that needed to violate the security level once, before the whole system is rendered vulnerable.

X.3.2 Micali_Schnorr_DRBG

The Micali_Schnorr_DRBG relies on the difficulty of factoring large integers for its security. Given a properly-generated RSA key (e,n), the DRBG can generate pseudorandom bits from an initial seed whose security is provably equivalent to the difficulty of factoring n. [[Nitpick: Is this using the same security assumption? Can I factor n if I can predict the output bits, or if I can distinguish them from random?]]

Note that if this DRBG is implemented in a way that uses and stores knowledge of the factors of n (e.g., using CRT), then the DRBG cannot achieve backtracking resistance. [[Is this right? Would it even make sense to use CRT for doing these exponentiations with some small, low-weight e?]]