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 three 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. All are believed to be secure at the time of this writing, and all are based ultimately on the strength of existing, widely-trusted cryptographic primitives.

D.2 HMAC_DRBG

HMAC_DRBG is a DRBG built around the use of some approved hash function in the HMAC construction. To generate pseudorandom bits from a secret key K and a starting value V, the DRBG computes

$$V = HMAC(K,V)$$

At the end of a generation request, the DRBG regenerates K and V, each requiring one HMAC computation.

Security. The security of HMAC_DRBG is based on the assumption that an approved hash function used in the HMAC construction is a pseudorandom function family. Informally, this just means that when an attacker doesn't know the key used, HMAC outputs look random even given knowledge and control over the inputs. In general, even relatively weak hash functions seem to be quite strong when used in the HMAC construction. On the other hand, there is not a reduction proof from the hash function's collision resistance properties to the security of the DRBG; the security of HMAC_DRBG depends on the security of the underlying hash function, but it is possible in principle for HMAC_DRBG to be broken by someone who cannot find collisions or preimages for the underlying hash function. That said, the pseudorandomness of HMAC is a widely used assumption in designing cryptographic protocols.

Performance. HMAC_DRBG produces pseudorandom outputs considerably more slowly than the underlying hash function processes inputs; for SHA256, a long generate request produces output bits at about 1/4 the rate the hash function can process input bits. Each generate request also involves additional overhead equivalent to processing 2048 extra bits with SHA256. Note, however, that hash functions are typically quite fast; few if any applications are expected to need output bits faster than HMAC_DRBG can provide them.

Resources. Any entropy source may be used with HMAC_DRBG, as it uses HMAC to process all its inputs. HMAC_DRBG requires access to a hashing engine or an HMAC implementation, and *2n* bits of additional storage space.

Algorithm Choices. At the time of this writing, there are five approved hash functions: SHA1, SHA224, SHA256, SHA384, and SHA512. SHA224 and SHA384 are simply variants of SHA256 and SHA512, respectively, with different starting values and truncation at the end; it typically makes no sense to use them in the HMAC_DRBG. SHA256 and SHA512 may be used to support any security level. SHA1 may be used only for the 112 and 128 bit security levels.

D.3 CTR_DRBG

CTR_DRBG is a DRBG based on running an approved block cipher in counter mode. At the time of this writing, only three-key TDEA and AES are approved for use within X9.82. Pseudorandom outputs are generated by encrypting successive values of a counter; after a generate request, a new key and starting counter value are generated.

Security. The security of CTR_DRBG is directly based on the security of the underlying block cipher, in the sense that so long as some limits on the total number of outputs are observed, any attack on CTR_DRBG represents an attack on the underlying block cipher.

Constraints on Outputs. For AES_CTR_DRBG, up to 2^{48} generate requests may be made, each of up to 2^{19} bits, with negligible chance of any weakness in AES_CTR_DRBG which does not represent a weakness in AES. However, the smaller block size of TDEA imposes more constraints: each generate request in TDEA_CTR_DRBG is limited to 2^{13} bits, and at most 2^{32} such requests may be made.

Performance. For large generate requests, CTR_DRBG produces outputs at the same speed as the underlying block cipher encrypts data. Further, CTR_DRBG is parallelizeable. At the end of each generate requests, work equivalent to between 2 and 4 block encryptions is done to derive new keys and counters for the next generation request.

Resources. CTR_DRBG is ideal for situations in which a conditioned entropy source is available or the entropy input for this DRBG is to come from another RBG. When this is not the case, a rather cumbersome derivation function must be used each time additional entropy or other input is provided. For instantiation and reseeding, however, this should not lead to an important performance penalty, since both these operations are done only very rarely. CTR_DRBG implementations also suffer a substantial performance penalty if they process additional input with generate requests. CTR_DRBG requires access to a block cipher engine, including the ability to change keys, and $n+k$ bits of storage, where n is the size of the cipher block in bits, and k is the size of its key.

Algorithm Choices. At the time of this writing, AES-128, AES-192, AES-256, and three-key TDEA (with 168 bit keys, but providing only about 112 bit strength) are permitted algorithms. The security level of the DRBG is the security level of the block cipher.

D.4 DRBGs Based on Hard Problems

The **Dual_EC_DRBG** bases its security on a "hard" number-theoretic problem. For the types of curves used in the **Dual_EC_DRBG**, the Elliptic Curve Discrete Logarithm Problem has no known attacks that are better than the "meet-in-the-middle" attacks, with a work factor of sqrt (2^m).

This algorithm is decidedly less efficient to implement than some of the others. However, in those cases where security is the utmost concern, as in SSL or IKE exchanges, the additional complexity is not usually an issue. Except for dedicated servers, time spent on the exchanges is just a small portion of the computational load; overall, there is no impact on throughput by using a number-theoretic algorithm. As for SSL or IPSEC servers, more and more of these servers are getting hardware support for cryptographic primitives like modular exponentiation and elliptic curve arithmetic for the protocols themselves. Thus, it makes sense to utilize those same primitives (in hardware or software) for the sake of high-security random numbers.

D.4.1 Implementation Considerations

Random bits are produced in blocks of bits representing the x-coordinates on an elliptic curve.

Because of the various security strengths allowed by this Standard there are multiple curves available, with differing block sizes. The size is always a multiple of 8, about 16 bits less than a curve's underlying field size. Blocks are concatenated and then truncated, if necessary, to fullfil a request for any number of bits up to a maximum per call of 10,000 times the block length. The smallest blocksize is 216, meaning that at least 2M bits can be requested on each call.)

An important detail concerning the Dual_EC_DRBG is that every call for random bits, whether it be for 2 million bits or a single bit, requires that at least one full block of bits be produced; no unused bits are saved internally from the previous call. Each block produced requires two point multiplications on an elliptic curve—a fair amount of computation. Applications such as IKE and SSL are encouraged to aggregate all their needs for random bits into a single call to Dual_EC_DRBG, and then parcel out the bits as required during the protocol exchange. A C structure, for example, is an ideal vehicle for this.

Comment [ebb1]: Page: 4
Doesn't this violate our guidance somewhere ?

To avoid unnecessarily complex implementations, it should be noted that *every* curve in the Standard need not be available to an application. To improve efficiency, there has been much research done on the implementation of elliptic curve arithmetic; descriptions and source code are available in the open literature.

As a final comment on the implementation of the Dual_EC_DRBG, note that having fixed base points offers a distinct advantage for optimization. Tables can be precomputed that allow nP to be attained as a series of point additions, resulting in an 8 to 10-fold speedup, or more, if space permits.