Appendix G: (Informative) DRBG Selection

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.

G.1 Hash_DRBG

Hash_DRBG is a DRBG based on using an approved hash function in a kind of counter mode. It is descended from the FIPS 186 DRBG. Each Generate request is met by starting a counter from the current secret state V and iterating it to generate each successive n bits of output requested, where n is the number of bits in the hash output.

At the end of the Generate request, the secret state V is updated in a way that prevents backtracking.

Performance. Within a Generate request, each n bits of output require one hash computation. This makes **Hash_DRBG** twice as fast as HMAC_DRBG. Each Generate request, one additional hash computation and some additions are done.

Security. **Hash_DRBG**'s security depends on the underlying hash function's behavior when processing a sequence of sequential integers. If the hash function were replaced by a random oracle, **Hash_DRBG** would be secure. It is difficult to relate the properties of the hash function required by **Hash_DRBG** with common properties such as collision resistance, preimage resistance, or pseudorandomness. There are known problems with **Hash_DRBG** when the DRBG is instantiated with insufficient entropy, and then provided enough entropy to reach a secure state via additional input to the Generate function.

Resources. **Hash_DRBG** requires access to a hashing engine, and the ability to do addition with *seedlen*-bit integers. **Hash_DRBG** makes extensive use of the hash-based derivation function, hash_df.

G.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 (*Key*) and a starting value *V*, the DRBG computes

$$V = \mathbf{HMAC}(Key, V).$$

At the end of a generation request, the DRBG regenerates *Key* 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 somewhat different properties of the underlying hash function. Note, however, that 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 SHA-256, a long generate request produces output bits at about 1/4 of the rate that the hash function can process input bits. Each generate request also involves additional overhead equivalent to processing 2048 extra bits with SHA-256. 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 input source may be used with **HMAC_DRBG**, as it uses HMAC to process all its inputs. **HMAC_DRBG** requires access to an HMAC implementation for optimal performance. However, a general-purpose hash implementation can always be used to implement HMAC. Any implementation requires the storage space required for the internal state (see Section 10.1.2.2.1).

Algorithm Choices. The choice of algorithms that may be used by **HMAC_DRBG** is discussed in Section 10.1.1.

G.3 CTR_DRBG

CTR_DRBG is a DRBG based on using an Approved block cipher in counter mode. At the time of this writing, only three-key TDEA and AES are approved for use within ANS X9.82. Pseudorandom outputs are generated by encrypting successive values of a counter; after a generate request, a new key and new 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 shown in Table 3 of Section 10.2.2.1, for each of the three AES key sizes, up to 2^{48} generate requests may be made, each of up to 2^{19} bits, with a negligible chance of any weakness that does not represent a weakness in AES. This tracks with the situation for most other DRBGs. However, the smaller block size of TDEA imposes more stringent constraints; each generate request 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. Furthermore, **CTR_DRBG** is parallelizeable. At the end of each generate request, work equivalent to between two and four block encryptions is done to derive new keys and counters for the next generation request.

Resources. **CTR_DRBG** may be implemented with or without a derivation function.

With a derivation function, **CTR_DRBG** can process additional inputs for Generate requests in the same way as any other DRBG, but at a cost in performance because of the use of the block cipher derivation function. Such an implementation may be seeded by any approved entropy source.

Without a derivation function, **CTR_DRBG** is more efficient, but less flexible. Such an implementation must be seeded by a conditioned entropy source or another RBG, and can accept additional input and personalization strings of less than *seedlen* bits.

CTR_DRBG requires access to a block cipher engine, including the ability to change keys, and the storage space required for the internal state (see Section 10.2.2.2.1).

Algorithm Choices. The choice of algorithms that may be used by **CTR_DRBG** is discussed in Section 10.2.1.

G.4 DRBGs Based on Hard Problems

[[I've rewritten this to be consistent in style with the rest of this section.]]

The **Dual_EC_DRBG** bases its security on a number-theoretic problem which is widely believed to be hard. 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).

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

Performance. 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 language structure, for example, is an ideal vehicle for this.

This algorithm is decidedly less efficient to implement than the other DRBGs. 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.

Constraints on Outputs.

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.)

Resources. The **Dual_EC_DRBG** implementation needs access to a hashing engine, and an engine for doing point multiplication on an elliptic curve. In addition, some integer arithmetic support is needed.

To avoid unnecessarily complex implementations, note 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.