November 28, 2007

Dear Bruce,

In your November 14, 2007 Wired commentary (http://www.wired.com/politics/security/commentary/securitymatters/2007/11/securitymatters_1115), you suggested that the Dual_EC_DRBG random number generator published in NIST Special Publication 800-90 has a property "that can only be described as a back door." We have no evidence that anyone has, or will ever have, the "secret numbers" for the back door that were hypothesized by mathematicians Dan Shumow and Neils Ferguson, that would provide advance information on the random numbers generated by the algorithm. For this reason, we are not withdrawing the algorithm at this time. NIST Special Publication 800-90, which includes a method for randomly generating points if there is a concern about a back door, underwent a rigorous review process that included a public comment period before it was published,. All NIST algorithms, including the Dual_EC_DRBG, undergo continual review throughout their lifetime. If successful attacks are found on an algorithm, the algorithm is withdrawn.

Sincerely,

Elaine Barker
Lead Author, NIST SP 800-90
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD

A.2.1 Generating Alternative *P,Q*

The curve **shall** be one of the NIST curves from FIPS 186-3 that is specified in Appendix A.1 of this Recommendation, and **shall** be appropriate for the desired *security_strength*, as specified in Table 4, Section 10.3.1.

The points P and Q **shall** be valid base points for the selected elliptic curve that are generated to be verifiably random using the procedure specified in ANS X9.62. The following input is required for each point:

> An elliptic curve $E = (F_p, a, b)$, cofactor h, prime n, a bit string *domain_parameter_seed*[1], and hash function **Hash**(). The curve parameters are given in Appendix A.1 of this Recommendation. The *domain_parameter_seed* **shall** be different for each point, and the minimum length m of each *domain_parameter_seed* **shall** conform to Section 10.3.1, Table 4, under "Seed length". The bit length of *domain_parameter_seed* may be larger than m. The hash function **shall** be SHA-512 in all cases.

The *domain_parameter_seed* **shall** be different for each point P and Q. A domain parameter seed **shall not** be the seed used to instantiate a DRBG. The domain parameter seed is an arbitrary value that may, for example, be determined from the output of a DRBG.

If the output from the ANS X9.62 generation procedure is "failure", a different *domain_parameter_seed* **shall** be used for the point being generated.

Otherwise, the output point from the generate procedure in ANS X9.62 **shall** be used.

[EBB: Does this take care of the required relationship $Q = aP$ that is stated in Section 10.3?]

A.2.2 Additional Self-testing Required for Alternative *P,Q*

To insure that the points P and Q have been generated appropriately, additional self-test procedures **shall** be performed whenever the instantiate function is invoked. Section 11.3.1 specifies that known-answer tests on the instantiate function be performed prior to creating an operational instantiation. As part of these tests, an implementation of the generation procedure in ANS X9.62 **shall** be called for each point (i.e., P and Q) with the appropriate *domain_parameter_seed* value that was used to generate that point. The point returned **shall** be compared with the corresponding stored value of the point. If the generated value does not match the stored value, the implementation **shall** halt with an error condition.

---

[1] Called a *SEED* in ANS X9.62.

E.1.4 Potential Bias Due to Modular Arithmetic for Curves Over F_p

Given an integer x in the range 0 to 2^N-1, the r^{th} bit of x depends solely upon whether $\left\lfloor \frac{x}{2^r} \right\rfloor$ is odd or even. If all of the values in this range are sampled uniformly, the r^{th} bit will be 0 exactly ½ of the time. But if x is restricted to F_P, i.e., to the range 0 to p-1, this statement is no longer true.

By excluding the $k = 2^N - p$ values p, p+1, ..., 2^N –1 from the set of all integers in Z_N, the ratio of ones and zeroes in the r^{th} bit is altered from $2^{N-1} / 2^{N-1}$ to a value that can be no smaller than $(2^{N-1} - k)/ 2^{N-1}$. For all the primes p used in this Recommendationt, $k/2^{N-1}$ is smaller than 2^{-31}. Thus, the ratio of ones and zeroes in any bit is within at least 2^{-31} of 1.0.

To detect this small difference from random, a sample of 2^{64} outputs is required before the observed distribution of 1's and 0's is more than one standard deviation away from flat random. This effect is dominated by the bias addressed below in section E.2.

Email note to interested parties:
A draft NIST Special Publication (Draft SP 800-90, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*) is available for public comment at http://csrc.nist.gov/publications/drafts.html. Comments should be submitted to ebarker@nist.gov by Wednesday, February 1, 2006. Please place "Comments on SP 800-90" in the subject line.

Instructions to Patrick:
Please place the following on the csrc page:

December 16, 2005: A draft NIST Special Publication (Draft SP 800-90, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*) is available[EBB1] for public comment. Comments should be submitted to ebarker@nist.gov by Wednesday, February 1, 2006. Please place "Comments on SP 800-90" in the subject line.

The draft document is attached.

Instructions to Larry Bassham:
Please place the following on the http://csrc.nist.gov/CryptoToolkit/tkrng.html page:

A draft NIST Special Publication (Draft SP 800-90, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*) is available[EBB2] for public comment. Comments should be submitted to ebarker@nist.gov by Wednesday, February 1, 2006. Please place "Comments on SP 800-90" in the subject line.

Patrick will be placing the document on the drafts page.

G.4 DRBGs Based on Hard Problems

The **Dual_EC_DRBG** generates pseudorandom outputs by extracting bits from elliptic curve points. The secret, internal state of the DRBG is a value S that is the x-coordinate of a point on an elliptic curve. Outputs are produced by first computing R to be the x-coordinate of the point $S*P$ and then extracting low order bits from the x-coordinate of the elliptic curve point $R*Q$.

Security. The security of **Dual_EC_DRBG** is based on the so-called "Elliptic Curve Discrete Logarithm Problem" that has no known attacks better than the so-called "meet-in-the-middle" attacks. For an elliptic curve defined over a field of size 2^m, the work factor of these attacks is approximately $2^{m/2}$ so that solving this problem is computationally infeasible for the curves in this document. The **Dual_EC_DRBG** is the only DRBG in this document whose security is related to a hard problem in Number Theory.

Constraints. For any one of the three elliptic curves, a particular instance of **Dual_EC_DRBG** may generate at most 2^{32} output blocks before reseeding. Since the sequence of output blocks are expected to cycle in approximately sqrt(n) bits (where n is the (prime) order of the particular elliptic curve being used), this is quite a conservative reseed interval for any one of the there possible curves.

Performance. Due to the elliptic curve arithmetic involved in this DRBG, this algorithm generates pseudorandom bits more slowly than the other DRBGS in this document. It should be noted, however, that the design of this algorithm allows for certain performance-enhancing possibilities. First, note that the use of fixed base points allows one to substantially increase the performance of this DRBG via the use of tables. By storing multiples of the points P and Q, the elliptic curve multiplication can be accomplished via point additions rather than multiplications, a much less expensive operation. In more constrained environments when table storage is not an option, the use of so-called Montgomery Coordinates of the form $(X : Z)$ can be used as a method to increase performance since the y-coordinates of the computed points are not required. A given implementation of this DRBG need not include all three of the NIST-approved curves. Once the designer decides upon the strength required by a given application, he can then choose to implement the single curve that most appropriately meets this requirement. For a common level of optimization expended, the higher strength curves will be slower and tend toward less efficient use of output blocks. To mitigate the latter, the designer should be aware that every distinct request for random bits, whether for two million bits or a single bit, requires the computational expense of at least two elliptic curve point multiplications. Applications requiring large blocks of random bits (such as IKE or SSL), can thus be implemented most efficiently by first making a single call to the DRBG for all the required bits and then appropriately partitioning these bits as required by the protocol. For applications that already have hardware or software support for elliptic curve arithmetic, this DRBG is a natural choice as it allows the designer to utilize existing capabilities to generate truly high-security random numbers.

Resources. Any entropy input source may be used with **Dual_EC_DRBG**, provided that it is capable of generating at least *seedlen* bits. This DRBG also requires an appropriate hash function (see Table 4) that is used exclusively for producing an appropriately-sized initial state from the entropy input at instantiation or reseeding. An implementation of this DRBG must also have enough storage for the internal state (see 10.3.1.1). Some optimizations require additional storage for moderate to large tables of pre-computed values.

Algorithm Choices. The choice of appropriate elliptic curves and points used by **Dual_EC_DRBG** is discussed in Appendix A.1.

Below are Table 4 from Section 10.3.1, and the example in Appendix F.5. I have some questions,

1. In the table, which entries are required for the curve to work properly, and which entries are really dependent on the requested security strength during instantiation? For example, if a consuming application requests instantiation at the 112-bit security level, theoretically, any curve could be used. If it is determined that the P-256 curve will be used in this case, do all the values in the P-256 column need to be used, or can lesser values be used in some cases.
    a. Can the *min_length*, for example, be reduced to 224? Why is the *min_length* about twice the minimum entropy anyway? To give a big security cushion?
    b. Can the *seedlen* be 224, as it was for the P-224 curve, or must it be 256?
    c. Can the *max_outlen* be 208, or must it be 240?

    These same questions apply if any curve is used to support a security strength < the max. that it can support?
2. Please check the blue text that is highlighted in gray. Does it make sense?

Table 4: Definitions for the Dual_EC_DRBG

	~~P-224~~ **P-256**	**P-384**	**P-521**
Supported security strengths	See SP 800-57		
highest_supported_ security_strength	See SP 800-57		
Output block length (*max_outlen* = largest multiple of 8 less than *seedlen* - (13 + $\log_2$ (the cofactor))	~~208~~ 240	368	504
Required minimum entropy for instantiate and reseed	*security_strength*		
Minimum entropy input length (*min _length* = 8 × $\lceil$ *seedlen*/8 $\rceil$)	~~224~~ 256	384	528
Maximum entropy input length (*max _length*)	$\leq 2^{13}$ bits		
Maximum personalization string length (*max_personalization_string_length*)	$\leq 2^{13}$ bits		
Supported security strengths	See SP 800-57		
Seed length (*seedlen* = *m*)	~~224~~ 256	384	521

	~~P-224~~ P-256	P-384	P-521
Appropriate hash functions	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512	SHA-224, SHA-256, SHA-384, SHA-512	SHA-256, SHA-384, SHA-512
max_number_of_bits_per_request	*max_outlen* × *reseed_interval*		
Number of blocks between reseeding (*reseed_interval*)	$\leq 2^{32}$ blocks		

~~The generate function is the same as that provided in Annex E.3.5.~~

F.5 Dual_EC_DRBG Example

This example of **Dual_EC_DRBG** allows a consuming application to instantiate using any of the threefour prime curves. The elliptic curve to be used is selected during instantiation in accordance with the following:

requested_instantiation_security_strength	Elliptic Curve
≤ 112	P-256
113 – 128	P-256
129 – 192	P-384
193 – 256	P-512

A reseed capability is available, but prediction resistance is not available. Both a *personalization_string* and an *additional_input* are allowed. A total of 10 internal states are provided. For this implementation, the algorithms are provided as inline code within the functions.

The nonce for instantiation (*instantiation_nonce*) consists of a random value with *security_strength*/2 bits of entropy; the nonce is obtained by a separate call to the **Get_entropy_input** routine than that used to obtain the entropy input itself.

The internal state contains values for *s*, *seedlen*, *p*, *a*, *b*, *n*, *P*, *Q*, ~~*r_old*~~, *block_counter* and *security_strength*. In accordance with Table 4 in Section 10.3.1, security strengths of 112, 128, 192 and 256 may be supported. SHA-256 has been selected as the hash function. The following definitions are applicable for the instantiate, reseed and generate functions:

1. *highest_supported_security_strength* = 256.
2. Output block length (*outlen*): See Table 4.
3. Required minimum entropy for the entropy input at instantiation and reseed = *security_strength*.

4. Minimum entropy input length (*min _length*): See Table 4.
5. Maximum entropy input length (*max _length*) = 1000 bits.
6. Maximum personalization string length (*max_personalization_string_length*) = 800 bits.
7. Maximum additional input length (*max_additional_input_length*) = 800 bits.
8. Seed length (*seedlen*): See Table 4.
9. Maximum number of bits per request (*max_number_of_bits_per_request*) = 1000 bits.
10. Reseed interval (*reseed_interval*) = ~~10,000~~2^{32} blocks.

F.5.1 Instantiation of Dual_EC_DRBG

This implementation will return a text message and an invalid state handle (-1) when an **ERROR** is encountered. **Hash_df** is specified in Section 10.4.1.

Instantiate_Dual_EC_DRBG (...):

Input: integer (*requested_instantiation_security_strength*), bitstring *personalization_string*.

Output: string *status*, integer *state_handle*.

Process:

Comment : Check the validity of the input parameters.

1. If (*requested_instantiation_security_strength* > 256) then **Return** ("Invalid *requested_instantiation_security_strength*", -1).
2. If (**len** (*personalization_string*) > 800), then **Return** ("*personalization_string* too long", -1).

Comment : Select the prime field curve in accordance with the *requested_instantiation_security_strength*

3. If *requested_instantiation_security_strength* ≤ 112), then

   {*security_strength* = 112; *seedlen* = 256; *outlen* = 240; *min_entropy_input_len* = 256}

   Else if (*requested_instantiation_security_strength* ≤ 128), then

   {*security_strength* = 128; *seedlen* = 256; *outlen* = 240; *min_entropy_input_len* = 256}

   Else if (*requested_instantiation_security_strength* ≤ 192), then

   {*security_strength* = 192;, *seedlen* = 384; *outlen* = 368; *min_entropy_input_len* = 384}

Else {*security_strength* = 256;, *seedlen* = 521; *outlen* = 504;
min_entropy_input_len = 528}.

4. Select the appropriate elliptic curve from Appendix A using the Table in Appendix F.5 to obtain the domain parameters *p*, *a*, *b*, *n*, *P*, and *Q*.

Comment: Request *entropy_input*.

5. (*status*, *entropy_input*) = **Get_entropy_input** (*security_strength*, *min_entropy_input_length*, 1000).

6. If (*status* ≠ "Success"), then **Return** ("~~Failure indication returned by~~Catastrophic failure of the *entropy_input* source:" || *status*, -1).

7. (*status*, *instantiation_nonce*) = **Get_entropy_input** (*security_strength*/2, *security_strength*/2, 1000).

8. If (*status* ≠ "Success"), then **Return** ("Catastrophic failure of ~~Failure indication returned by~~ the random nonce source:" || *status*, -1).

Comment: Perform the instantiate algorithm.

9. *seed_material* = *entropy_input* || *instantiation_nonce* || *personalization_string*.

10. *s* = **Hash_df** (*seed_material*, *seedlen*).

11. ~~*r_old* = φ(*x*(*s* * *Q*)).~~

~~12.~~ *block_counter* = 0.

Comment: Find an unused internal state and save the initial values.

~~13~~12. (*status*, *state_handle*) = **Find_state_space** ( ).

~~14~~13. If (*status* ≠ "Success"), then **Return** (*status*, -1).

~~15~~14. *internal_state* (*state_handle*) = {*s*, *seedlen*, *p*, *a*, *b*, *n*, *P*, *Q*, ~~*r_old*,~~ *block_counter*, *security_strength*}.

~~16~~15. **Return** ("Success", *state_handle*).

F.5.2 Reseeding a Dual_EC_DRBG Instantiation

The implementation is designed to return a text message as the status when an error is encountered.

Reseed_Dual_EC_DRBG_Instantiation (...):

Input: integer *state_handle*, string *additional_input_string*.

Output: string *status*.

Process:

Comment: Check the input parameters.

1. If ((*state_handle* < 0) or (*state_handle* > 9) or (*internal_state* (*state_handle*).*security_strength* = 0)), then **Return** ("State not available for the *state_handle*").
2. If (**len** (*additional_input*) > 800), then **Return** ("Additional_input too long").

   Comment: Get the appropriate *state* values for the indicated *state_handle.*
3. *s* = *internal_state* (*state_handle*).*s*, *seedlen* = *internal_state* (*state_handle*).*seedlen*, *security_strength* = *internal_state* (*state_handle*).*security_strength.*

   Comment: Request new *entropy_input* with the appropriate entropy and bit length.
3. (*status*, *entropy_input*) = **Get_entropy_input** (*security_strength*, *min_ entropy_input_length*, 1000).
4. If (*status* ≠ "Success"), then **Return** ("Catastrophic failure of ~~Failure indication returned by~~ the entropy source:"|| *status*).

   Comment: Perform the reseed algorithm.
5. *seed_material* = **pad8** (*s*) || *entropy_input* || *additional_input.*
6. *s* = **Hash_df** (*seed_material*, *seedlen*).

   Comment: Update the changed values in the *state.*
7. *internal_state* (*state_handle*).*s* = *s.*
8. *internal_state.block_counter* = 0.
9. **Return** ("Success").

F.5.3 Generating Pseudorandom Bits Using Dual_EC_DRBG

The implemenation returns a *Null* string as the pseudorandom bits if an error is encountered.

Dual_EC_DRBG (...):

Input: integer (*state_handle*, *requested_security_strength*, *requested_no_of_bits*), bitstring *additional_input*.

Output: string *status*, bitstring *pseudorandom_bits*.

Process:

Comment: Check for an invalid *state_handle*.

1. If ((*state_handle* < 0) or (*state_handle* > 9) or (*internal_state* (*state_handle*) = 0)), then **Return** ("State not available for the *state_handle*", *Null*).

Comment: Get the appropriate *state* values for the indicated *state_handle.*

2. *s* = *internal_state* (*state_handle*).*s*, *seedlen* = *internal_state* (*state_handle*).*seedlen*, *P* = *internal_state* (*state_handle*).*P*, *Q* = *internal_state* (*state_handle*).*Q*, ~~*r_old = internal_state (state_handle).r_old,*~~ *block_counter* = *internal_state* (*state_handle*).*block_counter.*

Comment: Check the rest of the input parameters.

3. If (*requested_number_of_bits* > 1000), then **Return** ("Too many bits requested", *Null*).
4. If (*requested_security_strength* > *security_strength*), then **Return** ("Invalid *requested_strength*", *Null*).
5. If (**len** (*additional_input*) > 800), then **Return** ("Additional_input too long", *Null*).

Comment: Check whether a reseed is required.

6. If $(block\_counter + \left\lceil \frac{requested\_number\_of\_bits}{outlen} \right\rceil > \underline{10,000}2^{32})$, then

   6.1 **Reseed_Dual_EC_DRBG_Instantiation** (*state_handle*, *additional_input*).

   6.2 If (*status* ≠ "Success"), then **Return** (*status*).

   6.3 *s* = *internal_state* (*state_handle*).*s*, *block_counter* = *internal_state* (*state_handle*).*block_counter.*

   6.4 *additional_input* = *Null.*

Comment: Execute the generate algorithm.

7. If (*additional_input* = *Null*) then *additional_input* = 0

Comment: *additional_input* set to *m* zeroes.

Else *additional_input* = **Hash_df** (**pad8** (*additional_input*), *seedlen*).

Comment: Produce *requested_no_of_bits*, *outlen* bits at a time:

8. *temp* = the *Null* string.
9. *i* = 0.
10. $t = s \oplus additional\_input.$
11. $s = \varphi(x(t * P))$.

12. $r = \varphi(x(s * Q))$.

13. ~~If ($r = r\_old$), then **Return** ("ERROR: outputs match", *Null*).~~

~~14. $r\_old = r$.~~

~~15.~~ $temp = temp \| (\textbf{rightmost}\ outlen\ \text{bits of}\ r)$.

~~16~~14. $additional\_input = 0^{seedlen}$. Comment: *seedlen* zeroes; *additional_input* is added only on the first iteration.

~~17~~15. $block\_counter = block\_counter + 1$.

~~18~~16. $i = i + 1$.

~~19~~17. If (**len** (*temp*) < *requested_no_of_bits*), then go to step 10.

~~20~~18. *pseudorandom_bits* = **Truncate** (*temp*, $i \times outlen$, *requested_no_of_bits*).

Comment: Update the changed values in the *state*.

~~21~~19. $internal\_state.s = s$.

~~22~~20. ~~$internal\_state.r\_old = r\_old$.~~

~~23.~~ $internal\_state.block\_counter = block\_counter$.

~~24~~21. **Return** ("Success", *pseudorandom_bits*).

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.

1. Overview: Constructing a DRBG from Algorithms and Entropy Sources

The rest of this document is primarily concerned with the algorithms for generating pseudorandom outputs and how they are to be implemented. The source of seeding material for the DRBGs is mostly left to the designer to get right. In this appendix, we briefly describe how this can be done.

2. Internally Seeded DRBG

The ideal situation for a full DRBG is to have ready access to some entropy source. The entropy source provides bit strings along with a promise about how much entropy the bit strings have. An example of an entropy source would be a ring oscillator sampled one hundred times per second, where extensive analysis had been done to ensure that each sequence of 100 bits sampled had at least 80 bits of entropy. Any DRBG with an internal source of entropy can be used to access its underlying entropy source: if the source DRBG promises k bits of security, then each new request for k or more bits of output with prediction resistance from the source DRBG can be assumed to contain k bits of min-entropy.

When the DRBG has an internal source of entropy, reseeding and instantiation can be done on demand, requests for prediction resistance can be honored, and when a DRBG hits a required reseed interval after having generated too many outputs, it can simply reseed.

An internally seeded DRBG may use a seedfile, as described below, but does not require one.

3. Externally Seeded DRBG

Many implementations of DRBGs will not have access to an entropy source. We call these externally seeded DRBGs. An externally seeded DRBG has the following requirements:

- The DRBG must be instantiated at a time when the DRBG has access to some entropy source, and the entropy provided for instantiation must be provided over a secure (private and authentic) channel. In some applications, the entropy source is only available during manufacture or device setup; in others, it is occasionally available (e.g., when a user is moving the mouse around on a laptop).
- The DRBG must maintain its working state for as long as the DRBG may be called upon to generate outputs. This typically requires some kind of persistent memory to avoid losing state during power down. This may be maintained directly as the state of the DRBG, or maintained in a seedfile, as described below.

Over time, an externally seeded DRBG may be able to accumulate entropy from additional inputs provided by the user or consuming application. For this reason, the DRBG implementation should accept additional input whenever possible.

Implementations that have values which may have entropy, such as timestamps, nonces from protocol runs, etc., should provide them to the DRBG as additional inputs.

4. Using a Persistent DRBG as a Seedfile

A seedfile is persistent storage kept for a DRBG. It is used both to protect against silent failure of its entropy source, and also to allow externally seeded DRBGs to instantiate itself on power up and save all the entropy in its state back to the seedfile whenever necessary. Seedfiles are used and described in many cryptographic PRNGs, including /dev/random and Yarrow-160.

In X9.82, a seedfile is simply a DRBG instance whose working state is stored in some kind of persistent storage. Let SEEDFILE be the DRBG which is being used as a seedfile, and whose working state is stored in persistent storage. The following explains how the seedfile is used to support a DRBG (CURRENT) whose state is stored in volatile storage, but which may accumulate entropy over time from additional inputs:

- SEEDFILE is instantiated from some entropy source (possibly another RBG) when it is available, such as during manufacturing or device setup.
- At power up, CURRENT is instantiated. This is done by requesting a seed from SEEDFILE's generate routine and using that seed to instantiate CURRENT. Any additional input which is available from the application should be provided in the personalization string during CURRENT's instantiation.
- During operation, application data which might contain some entropy should be stored up, and periodically used as additional input in one-byte generate requests to SEEDFILE. The resulting one-byte outputs are discarded.
- At power down, CURRENT generates a k bit output. This output is used as additional input, along with any other available application data which might have some entropy, in a one-byte generate request to SEEDFILE. The one byte output is discarded.
- If the application rarely or never has a power down, then a k-bit value from CURRENT should periodically (e.g., once a day) be generated and used as additional input in a one-byte generate request to SEEDFILE, and the resulting output byte should be discarded.

5. Seeding Many DRBGs from One DRBG

Some applications may benefit from using different DRBGs for different applications. However, this must be done with some care, to avoid introducing new weaknesses. In order to use multiple different DRBGs for different consuming applications, the following steps shall be done:

- The parent DRBG is first instantiated with as much entropy as is available.
- Each child DRBG is instantiated with seed material acquired from a generate request to the parent DRBG.

Dear Dr. Schneier,

In light of your November 14, 2007 Wired commentary (http://www.wired.com/politics/security/commentary/securitymatters/2007/11/securitymatters_1115), we would like to take the opportunity to provide a few clarifications on NIST Special Publication 800-90.

NIST would never knowingly support the inclusion of an algorithm with secret features such as a "back door" in its standards. We do not think there is an intentionally placed back door or any other secret feature in the Dual_EC_DRBG pseudorandom-number generator.

If we discovered a back door in any algorithm in a NIST standard, we would withdraw the algorithm as soon as practical. We have no evidence that someone knows the existence of the "secret numbers" that Dan Shumow and Niels Ferguson have shown would provide advance information about the pseudorandom numbers that Dual_EC_DRBG would generate. Therefore, we have no plans to withdraw the algorithm at this time.

As you note, the Dual_EC_DRBG algorithm has also been approved as an ANSI international standard. The algorithm was vetted through the ANSI X9 subcommittee, of which Neils Ferguson (one of authors of the paper that claims a back door) is[EBB1] a participant. As Drs. Shumow and Ferguson state in their presentation, they do not believe that NIST would have intentionally created a back door in Dual_EC_DRBG, and they state that even the algorithm's designer may not have been aware of having potentially created such a feature.

It is also worth noting that no one is required to use Dual_EC_DRBG or any other algorithm based on its appearance in NIST Special Publication 800-90. Moreover, as you point out in your column, Appendix A of SP 800-90 gives users the information that is needed to generate alternative values which should preclude any chance of the secret trap door in the scenario that Shumow and Ferguson have presented.

NIST special publications, including this one, undergo a rigorous review process, including a public comment period. We take all comments on our publications very seriously and regularly update topics in our special publications. We appreciate the opportunity to comment on this standard[EBB2].

Sincerely,