NIST Special Publication 800-90

DRAFT (~~September~~December ~~22,~~ 2005)

Recommendation for Random Number Generation Using Deterministic Random Bit Generators

Elaine Barker and John Kelsey

National Institute of Standards and Technology

COMPUTER SECURITY

Abstract

This Recommendation specifies mechanisms for the generation of random bits using deterministic methods. The methods provided are based on either hash functions, block cipher algorithms or number theoretic problems.

KEY WORDS: deterministic random bit generator (DRBG); entropy; hash function; random number generator

Table of Contents

~~Contents~~

Random Number Generation Using Deterministic Random Bit Generators

1 Authority

This document has been developed by the National Institute of Standards and Technology (NIST) in furtherance of its statutory responsibilities under the Federal Information Security Management Act (FISMA) of 2002, Public Law 107-347.

NIST is responsible for developing standards and guidelines, including minimum requirements, for providing adequate information security for all agency operations and assets, but such standards and guidelines **shall not** apply to national security systems. This recommendation is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130, Section 8b(3), Securing Agency Information Systems, as analyzed in A-130, Appendix IV: Analysis of Key Sections. Supplemental information is provided in A-130, Appendix III.

This recommendation has been prepared for use by Federal agencies. It may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright. (Attribution would be appreciated by NIST.)

Nothing in this Recommendation should be taken to contradict standards and guidelines made mandatory and binding on federal agencies by the Secretary of Commerce under statutory authority. Nor should this Recommendation be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other federal official.

Conformance testing for implementations of the deterministic random bit generators (DRBGs) that are specified in this Recommendation will be conducted within the framework of the Cryptographic Module Validation Program (CMVP), a joint effort of NIST and the Communications Security Establishment of the Government of Canada. An implementation of a DRBG must adhere to the requirements in this Recommendation in order to be validated under the CMVP. The requirements of this Recommendation are indicated by the word "shall."

2 Introduction

This Recommendation specifies techniques for the generation of random bits that may then be used directly or converted to random numbers when random values are required by applications using cryptography.

There are two fundamentally different strategies for generating random bits. One strategy is to produce bits non-deterministically, where every bit of output is based on a physical process that is unpredictable; this class of random bit generators (RBGs) is commonly known as non-

deterministic random bit generators (NRBGs)[1]. The other strategy is to compute bits deterministically using an algorithm; this class of RBGs is known as Deterministic Random Bit Generators (DRBGs)[2]. This Recommendation will specify Approved DRBG mechanisms.

A DRBG uses an algorithm that produces a sequence of bits from an initial value that is determined by a seed. Once the seed is provided and the initial value determined, the DRBG is said to be instantiated. Because of the deterministic nature of the process, a DRBG is said to produce pseudorandom bits, rather than random bits. The seed used to instantiate the DRBG must contain sufficient entropy to provide assurance of randomness. If the seed is kept secret, and the algorithm is well designed, the bits output by the DRBG will ~~appear to be random~~be unpredictable, up to the security strength of the DRBG algorithm. However, the security provided by an RBG that uses a DRBG is a system implementation issue; both the DRBG and its source of entropy must be considered when determining whether the RBG is appropriate for use by consuming applications. Therefore, in this Recommendation the acronym RBG will be used to mean a DRBG~~, together with~~ and its source of entropy.

3 Scope

This Recommendation includes:

1. Requirements for the use of deterministic random bit generator mechanisms,
2. Specifications for deterministic random bit generator mechanisms that use hash functions, block ciphers and number theoretic problems,
3. Implementation issues, and
4. Assurance considerations.

This Recommendation specifies several diverse DRBG mechanisms, all of which provided acceptable security when this Recommendation was published. However, in the event that new attacks are found on a particular class of DRBG mechanisms, a diversity of approved mechanisms will allow a timely transition to a different class of DRBG mechanism.

Random number generation does not require interoperability between two entities, e.g., communicating entities may use different DRBG mechanisms without affecting their ability to communicate. Therefore, an entity may choose a single appropriate DRBG mechanism for their consuming applications; see Annex ~~D~~ G for a discussion of DRBG selection.

The precise structure, design and development of a random bit generator is outside the scope of this Recommendation.

---

[1] NRBGs have also been called True Random Number (or Bit) Generators or Hardware Random Number Generators.

[2] DRBGS have also been called Pseudorandom Bit Generators.

This Recommendation provides preliminary guidance on the selection of an entropy source and the construction of an RBG from an entropy source and an Approved DRBG. Additional guidance is under development in these areas.

4 Terms and Definitions

For the purposes of this part of the Recommendation, the following terms and definitions apply.

Algorithm	A clearly specified mathematical process for computation; a set of rules that, if followed, will give a prescribed result.
Approved	FIPS approved ~~or~~, NIST Recommended and/or validated by the Cryptographic Module Validation Program (CMVP). ~~An algorithm or technique that is either 1) specified in a FIPS or NIST Recommendation, or 2) adopted in a FIPS or NIST Recommendation and specified either (a) in an appendix to the FIPS or NIST Recommendation, or (b) in a document referenced by the FIPS or NIST Recommendation~~
Backtracking Resistance	The assurance that the output sequence from an RBG remains indistinguishable from an ideal random sequence even to an attacker who compromises the RBG in the future, up to the claimed security strength of the RBG. For example, an RBG that allowed an attacker to "backtrack" from the current working state to generate prior outputs would not provide backtracking resistance. The complementary assurance is called Prediction Resistance.
Biased	A value that is chosen from a sample space is said to be biased if one value is more likely to be chosen than another value. Contrast with unbiased.
Bitstring	A bitstring is an ordered sequence of 0's and 1's. The leftmost bit is the most significant bit of the string and is the newest bit generated. The rightmost bit is the least significant bit of the string.
Bitwise Exclusive-Or	An operation on two bitstrings of equal length that combines corresponding bits of each bitstring using an exclusive-or operation.
Block Cipher	A symmetric key cryptographic algorithm that transforms a block of information at a time using a cryptographic key. For a block cipher algorithm, the length of the input block is the same as the length of the output block.

Consuming Application	The application (including middle ware) that uses random numbers or bits obtained from an Approved random bit generator.
Cryptographic Key (Key)	A parameter that determines the operation of a cryptographic function such as: 1. The transformation from plaintexttociphertext and vice versa, 2. The synchronized generation of keying material, 3. A digital signature computation or verification.
Deterministic Algorithm	An algorithm that, given the same inputs, always produces the same outputs.
Deterministic Random Bit Generator (DRBG)	An RBG that uses a deterministic algorithm to produce a pseudorandom sequence of bits from a secret initial value called a *seed* along with other possible inputs. A DRBG is often called a Pseudorandom Number (or Bit) Generator.
DRBG Boundary	A conceptual boundary that is used to explain the operations of a DRBG and its interaction with and relation to other processes.
Entropy	A measure of the disorder, randomness or variability in a closed system. The entropy of X is a mathematical measure of the amount of information provided by an observation of X. As such, entropy is always relative to an observer and his or her knowledge prior to an observation. Also, see min-entropy.
Entropy Input	The input to an RBG of a string of bits that contains entropy, that is, the entropy input is digitized and is assessed. For an NRBG, this is obtained from an entropy source. For a DRBG, this is included in the seed material.
Entropy Source	A source of unpredictable data. There is no assumption that the unpredictable data has a uniform distribution. The entropy source includes a noise source, such as thermal noise or hard drive seek times; a digitalization process; an assessment process; an optional conditioning process and health tests. Thus, the entropy source provides bitstrings containing entropy and an assessment of the entropy that is provided.
Equivalent Process	Two processes are equivalent if, when the same values are input to each process, the same output is produced.

Exclusive-or	A mathematical operation, symbol ⊕, defined as: 0 ⊕ 0 = 0 0 ⊕ 1 = 1 1 ⊕ 0 = 1 1 ⊕ 1 = 0. Equivalent to binary addition without carry.
Full Entropy	Each bit of a bitstring with full entropy is unpredictable (with a uniform distribution) and independent of every other bit of that bitstring. ~~An *m* bit string has full entropy if every *m* bit value is equally likely to occur.~~
Hash Function	A (mathematical) function that maps values from a large (possibly very large) domain into a smaller range. The function satisfies the following properties: 1. (One-way) It is computationally infeasible to find any input that maps to any pre-specified output; 2. (Collision free) It is computationally infeasible to find any two distinct inputs that map to the same output.
Health Testing	Testing within an implementation immediately prior to or during normal operation to determine that the implementation continues to perform as implemented and as validated (if implementation validation was performed).
Implementation	An implementation of an RBG is a cryptographic device or portion of a cryptographic device that is the physical embodiment of the RBG design, for example, some code running on a computing platform.
Implementation Testing for Validation	Testing by an independent and accredited party to ensure that an implemention of this Recommendation conforms to the specifications of this Recommendation.
Instantiation of an RBG	An instantiation of an RBG is a specific, logically independent, initialized RBG. One instantiation is distinguished from another by a handle (e.g., an identifying number).
Internal State	The collection of stored information about an RBG instantiation. This can include both secret and non-secret information.
Key	See Cryptographic Key.

Min-entropy	The worst-case (i.e., the greatest lower bound) measure of uncertainty for a random variable.
Non-Deterministic Random Bit Generator (Non-deterministic RBG) (NRBG)	An RBG that produces output that is fully dependent on some unpredictable physical source that produces entropy. Contrast with a DRBG. Other names for non-deterministic RBGs are True Random Number (or Bit) Generators and, simply, Random Number (or Bit) Generators.
Personalization String	An optional string of bits that is combined with a secret input and a nonce to produce a seed.
Prediction Resistance	~~A~~ Assurance that a compromise of the DRBG internal state has no effect on the security of future DRBG outputs. If a compromise of state occurs, prediction resistance provides assurance that the output sequence resulting from state after the compromise remains secure. That is, an adversary who is given access to all of any subset of the output sequence after the compromise cannot distinguish it from random; if the adversary knows only part of the future output sequence, an adversary he cannot predict any bit of that future output sequence that he has not already seen. The complementary assurance is called Backtracking Resistance.
Pseudorandom	A process (or data produced by a process) is said to be pseudorandom when the outcome is deterministic, yet also effectively random as long as the internal action of the process is hidden from observation. For cryptographic purposes, "effectively" means "within the limits of the intended cryptographic strength."
Pseudorandom Number Generator	See Deterministic Random Bit Generator.
Public Key	In an asymmetric (public) key cryptosystem, that key of an entity's key pair that is publicly known.
Public Key Pair	In an asymmetric (public) key cryposystem, the public key and associated private key.
Random Number	For the purposes of this Recommendation, a value in a set that has an equal probability of being selected from the total population of possibilities and, hence, is unpredictable. A random number is an instance of an unbiased random variable, that is, the output produced by a uniformly distributed random process.

Random Bit Generator (RBG)	A device or algorithm that outputs a sequence of binary bits that appears to be statistically independent and unbiased.
Random Number Generator (RNG)	A device or algorithm that can produce a sequence of -random numbers that appears to be from an ideal random distribution.
Reseed	To aquire additional bits with sufficient entropy for the desired security strength
Security Strength	A number associated with the amount of work (that is, the number of operations) that is required to break a cryptographic algorithm or system; a security strength is specified in bits and is a specific value from the set (112, 128, 192, 256). The amount of work needed is $2^{security\_strength}$.
Seed	Noun : A string of bits that is used as input to a Deterministic Random Bit Generator (DRBG). The seed will determine a portion of the internal state of the DRBG, and its entropy must be sufficient to support the security strength of the DRBG. Verb : To aquire bits with sufficient entropy for the desired security strength. These bits will be used as input to a DRBG to determine a portion of the initial internal state. Also see reseed.
Seedlife	The length of the seed period.
Seed Period	The period of time between initializing a DRBG with one seed and reseeding that DRBG with another seed.
Sequence	An ordered set of quantities.
Shall	Used to indicate a requirement of this Recommendation.
Should	Used to indicate a highly desirable feature for a DRBG that is not necessarily required by this Recommendation.
String	See Bitstring.
Unbiased	A value that is chosen from a sample space is said to be unbiased if all potential values have the same probability of being chosen. Contrast with biased.

Unpredictable	In the context of random bit generation, an output bit is unpredictable if an adversary has only a negligible advantage (that is, essentially not much better than chance) in predicting it correctly.
Working State	A subset of the internal state that is used by a DRBG to produce pseudorandom bits at a given point in time. The working state (and thus, the internal state) is updated to the next state prior to producing another string of pseudorandom bits.

5 Symbols and Abbreviated Terms

The following abbreviations are used in this document:

Abbreviation	Meaning
AES	Advanced Encryption Standard.
DRBG	Deterministic Random Bit Generator.
ECDLP	Elliptic Curve Discrete Logarithm Problem.
FIPS	Federal Information Processing Standard.
HMAC	Keyed-Hash Message Authentication Code.
NRBG	Non-deterministic Random Bit Generator.
RBG	Random Bit Generator.
TDEA	Triple Data Encryption Algorithm.

The following symbols are used in this document.

Symbol	Meaning
+	Addition
$\lceil X \rceil$	Ceiling: the smallest integer $\geq X$. For example, $\lceil 5 \rceil = 5$, and $\lceil 5.3 \rceil = 6$.
$\lfloor X \rfloor$	The largest integer less than or equal to X. For example, $\lfloor 5 \rfloor = 5$, and $\lfloor 5.3 \rfloor = 5$.
$X \oplus Y$	Bitwise exclusive-or (also bitwise addition mod 2) of two bitstrings X and Y of the same length.
$X \| Y$	Concatenation of two strings X and Y. X and Y are either both bitstrings, or both octet strings.
gcd (x, y)	The greatest common divisor of the integers x and y.
len (a)	The length in bits of string a.

Symbol	Meaning
x **mod** n	The unique remainder r (where $0 \leq r \leq n$-1) when integer x is divided by n. For example, 23 mod 7 = 2.
	Used in a figure to illustrate a "switch" between sources of input.
$\{a_1, ...a_i\}$	The internal state of the DRBG at a point in time. The types and number of the a_i depends on the specific DRBG.
0^x	A string of x zero bits.

6 Document Organization

This Recommendation is organized as follows:

— Section 7 provides a functional model for a DRBG and discusses the major DRBG components.

— Section 8 provides ~~DRBG~~ concepts and general requirements~~. This section provides concepts and general requirements~~ for the implementation and use of a DRBG. ~~The DRBG functions are explained and requirements for an implementation are provided.~~

— Section 9 specifies the DRBG functions introduced in Section 8. These functions use the DRBG algorithms specified in Section 10.

— Section 10 specifies Approved DRBG algorithms. Algorithms have been specified that are based on the hash functions specified in FIPS 180-2 (Secure Hash Standard), block cipher algorithms specified in FIPS 197 and NIST Special Publication 800-67 (AES and TDEA, respectively), and a number theoretic problem that is expressed in elliptic curve technology.

— Section 11 addresses assurance issues for DRBGs, including documentation requirements, implementation validation and health testing,

This Recommendation also includes the following appendices:

— Appendix A specifies additional DRBG-specific information.

— Appendix B provides conversion routines.

— Appendix C provides guidance on entropy and entropy sources.

— Appendix D provides guidance on the construction of a random bit generator from an entropy source and a DRBG.

— Appendix E discusses security considerations ~~for implementing DRBGs~~when extracting bits in the Dual_EC_DRBG.

— Appendix F provides example pseudocode for each DRBG.

— Appendix G provides a discussion on DRBG selection.

— Appendix H provides references.

7 DRBG Functional Model

Figure 1 provides a functional model of DRBGs. The components of this model are discussed in the following subsections.

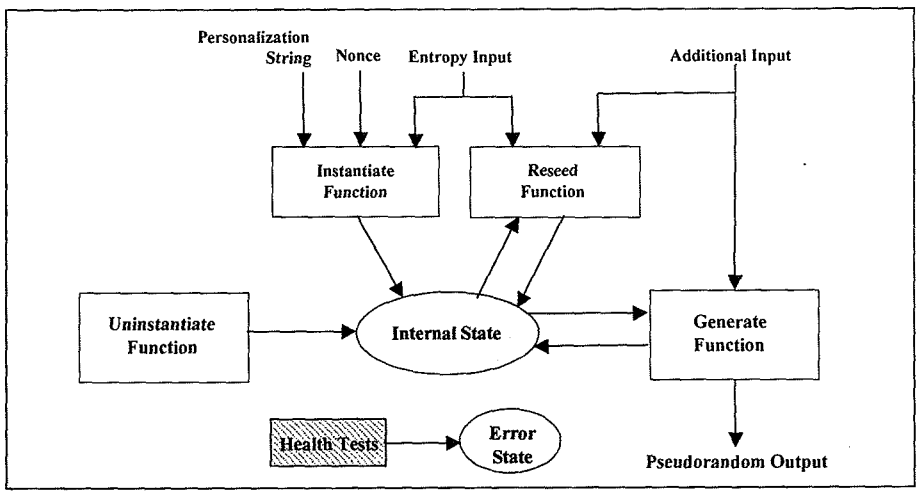

Figure 1: DRBG Functional Model

7.2~~1~~ Entropy Input

The entropy input is provided to a DRBG for the seed (see Section 8.6). The entropy input and the seed **shall** be kept secret. The secrecy of this information provides the basis for the security of the DRBG. At a minimum, the entropy input **shall** provide the requested amount of entropy for a DRBG. Appropriate sources for the entropy input are discussed in Appendix C.

Ideally, the entropy input will be full entropy; however, the DRBGs have been specified to allow for some bias in the entropy input by allowing the length of the entropy input to be longer than the required amount of entropy (expressed in bits). The entropy input can be defined to be a variable length (within limits), as well as fixed length. In all cases, the DRBG expects that when entroy input is requested, the returned bitstring will contain at least the requested amount of entropy. ~~The DRBGs allow for some bias in the entropy input. Whenever a bitstring containing entropy is required by the DRBG, a request is made that indicates the minimum amount of entropy to be returned; the request may obtain entopy input bits from a buffer containing readily available entropy bits or may cause entropy input bits to be acquired. The request may be fulfilled by a bitstring that is equal to or greater in length than the requested entropy. The DRBG expects that the returned bitstring will contain at least the amount of entropy requested.~~ Additional entropy beyond

the amount requested is not required, but is desirable.

7.3~~2~~ Other Inputs

Other information may be obtained by a DRBG as input. This information may or may not be required to be kept secret by a consuming application; however, the security of the DRBG itself does not rely on the secrecy of this information. The information **should** be checked for validity when possible.

During DRBG instantiation, a nonce ~~is~~ may be required, and if used, it is combined with the entropy input to create the initial DRBG seed. ~~Criteria for the~~The nonce and its use are ~~provided~~ discussed in Sections 8.6.1 and ~~8.6.7~~.

This Recommendation strongly advi~~s~~zes the insertion of a personalization string during DRBG instantiation; when used, the personalization string is combined with the entropy bits and a nonce to create the initial DRBG seed. The personalization string **shall** be unique for all instantiations of the same DRBG type (e.g., **HMAC_DRBG**). See Section 8.7.1 for additional discussion on personalization strings.

Additional input may also be provided during reseeding and when pseudorandom bits are requested. See Section 8.7.2 for a discussion of this input.

7.4~~3~~ The Internal State

The internal state is the memory of the DRBG and consists of all of the parameters, variables and other stored values that the DRBG uses or acts upon. The internal state contains both administrative data (e.g., the security level) and data that is acted upon and/or modified during the generation of pseudorandom bits (i.e., the *working state*). The contents of the internal state is dependent on the specific DRBG and includes all information that is required to produce the pseudorandom bits from one request to the next.

7.5~~4~~ The DRBG Functions

The DRBG functions handle the DRBG's internal state. The DRBGs in this Recommendation have four separate functions (exclusive of health tests):

1. The instantiate function acquires entropy input and may combine it with a nonce and a personalization string to create a seed from which the initial internal state is created.
2. The generate function generates pseudorandom bits upon request, using the current internal state, and generates a new internal state for the next request.
3. The reseed function acquires new entropy input and combines it with the current internal state and any additional input that is provided to create a new seed and a new internal state.
4. The uninstantiate function zeroizes (i.e., erases) the internal state.

7.6~~5~~ Health Tests

Health testing is used to determine that the DRBG continues to function correctly. The health tests are discussed in Sections 9.5 and 11.4~~3~~.

8. DRBG Concepts and General Requirements

8.1 DRBG Functions

A DRBG requires instantiate, uninstantiate, generate, and testing functions. A DRBG maymay also include a reseed function. A DRBG **shall** be instantiated prior to the generation of output by the DRBG. These functions are specified in Section 9.

8.2 DRBG Instantiations

A DRBG maymay be used to obtain pseudorandom bits for different purposes (e.g., DSA private keys and AES keys) and maymay be separately instantiated for each purpose.

A DRBG is instantiated using a seed and maymay be reseeded; when reseeded, the seed **shall** be different than the seed used for instantiation. Each seed defines a *seed period* for the DRBG instantiation; an instantiation consists of one or more seed periods that begin when a new seed is acquired (see Figure 2).

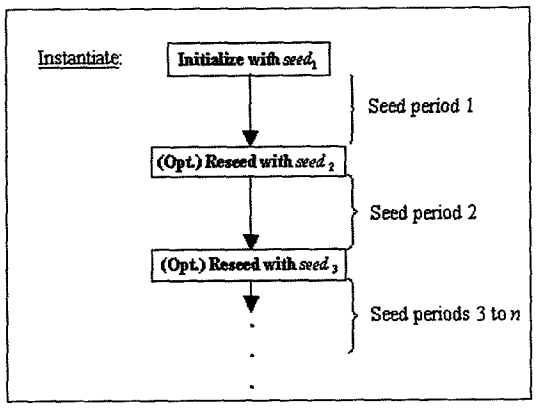

Figure 2: DRBG Instantiation

8.3 Internal States

During instantiation, an initial internal state is derived from the seed. The internal state for an instantiation includes:

1. Working state:
   a. One or more values that are derived from the seed and become part of the internal state; these values must usually remain secret, and
   b. A count of the number of requests or blocks produced since the instantiation was seeded or reseeded.
2. Administrative information (e.g., security strength and prediction resistance flag).

The internal state **shall** be protected at least as well as the intended use of the pseudorandom output bits requested by the consuming application. A DRBG implementation maymay be designed to handle multiple instantiations. Each DRBG instantiation **shall** have its own internal state. The internal state for one DRBG instantiation **shall not** be used as the internal state for a different instantiation.

A DRBG transitions between internal states when the generator is requested to provide

new pseudorandom bits. A DRBG maymay also be implemented to transition in response to internal or external events (e.g., system interrupts) or to transition continuously (e.g., whenever time is available to run the generator).

8.4 Security Strengths Supported by an Instantiation

The DRBGs specified in this Recommendation support four security strengths: 112, 128, 192 or 256 bits. The actual security strength supported by a given instantiation depends on the DRBG implementation and on the amount of entropy provided to the instantiate function. Note that the security strength actually supported by a particular instantiation maycould be less than the maximum security strength possible for that DRBG implementation (see Table 1). For example, a DRBG that is designed to support a maximum security strength of 256 bits could be instantiated to support only a 128-bit security strength if the additional security provided by the 256-bit security strength is not required.

Table 1: Possible Instantiated Security Strengths

Maximum Designed Security Strength	**112**	**128**	**192**	**256**
Possible Instantiated Security Strengths	112	112, 128	112, 128, 192	112, 128, 192, 256

A security strength for the instantiation is requested by a consuming application during instantiation, and the instantiate function obtains the appropriate amount of entropy for the requested security strength. Any security strength may be requested, but the DRBG will only be instantiated to one of the four security strengths above, depending on the DRBG implementation. A requested security strength that is below the 112-bit security strength or is between two of the four security strengths will be instantiated to the next highest level (e.g., a requested security strength of 96 bits will result in an instantiation at the 112-bit security strength).

Following instantiation, requests can be made to the generate function for pseudorandom bits. For each generate request, a security strength to be provided for the bits is requested. Any security strength can be requested up to the security strength of the instantiation, e.g., an instantiation could be instantiated at the 128-bit security strength, but a request for pseudorandom bits could indicate that a lesser security strength is actually required for the bits to be generated. The generate function checks that the requested security strength does not exceed the security strength for the instantiation. Assuming that the request is valid, the requested number of bits is returned.

When an instantiation is used for multiple purposes, the minimum entropy requirement for each purpose must be considered. The DRBG needs to be instantiated for the highest security strength required. For example, if one purpose requires a security strength of 112 bits, and another purpose requires a security strength of 256 bits, then the DRBG needs to

be instantiated to support the 256-bit security strength.

8.5 DRBG Boundaries

As a convenience, this Recommendation uses the notion of a "DRBG boundary" to explain the operations of a DRBG and its interaction with and relation to other processes; a DRBG boundary contains all DRBG functions and internal states required for a DRBG. A DRBG boundary is entered via the DRBG's public interfaces, which are made available to consuming applications.

Within a DRBG boundary,

1. The DRBG internal state and the operation of the DRBG functions **shall** only be affected according to the DRBG specification.
2. The DRBG internal state **shall** exist solely within the DRBG boundary. The internal state **shall** be contained within the DRBG boundary and **shall not** be accessed by non-DRBG functions or other instantiations of that or other DRBGs.
3. Information about secret parts of the DRBG internal state and intermediate values in computations involving these secret parts **shall not** affect any information that leaves the DRBG boundary, except as specified for the DRBG pseudorandom bit outputs.

Each DRBG includes one or more cryptographic primitives (e.g., a hash function). Other applications may use the same cryptographic primitive as long as the DRBG's internal state and the DRBG functions are not affected.

A DRBG's functions may be contained within a single device, or may be distributed across multiple devices (see Figures 3 and 4). Figure 3 depicts a DRBG for which all functions are contained within the same device. Figure 4 provides an example of DRBG functions that are distributed across multiple devices. In this latter case, each device has a DRBG sub-boundary that contains the DRBG functions implemented on that device, and the boundary around the entire DRBG consists of the aggregation of sub-boundaries providing the DRBG functionality. The use of distributed DRBG functions may be convenient for restricted environments (e.g., smart card applications) in which the primary use of the DRBG does not require repeated use of the instantiate or reseed functions.

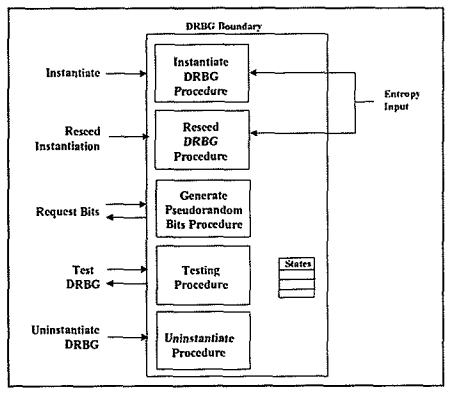

Figure 3: DRBG Functions within a Single Device

Although the entropy input is shown in the figures as originating outside the DRBG

boundary, it may originate from within the boundary.

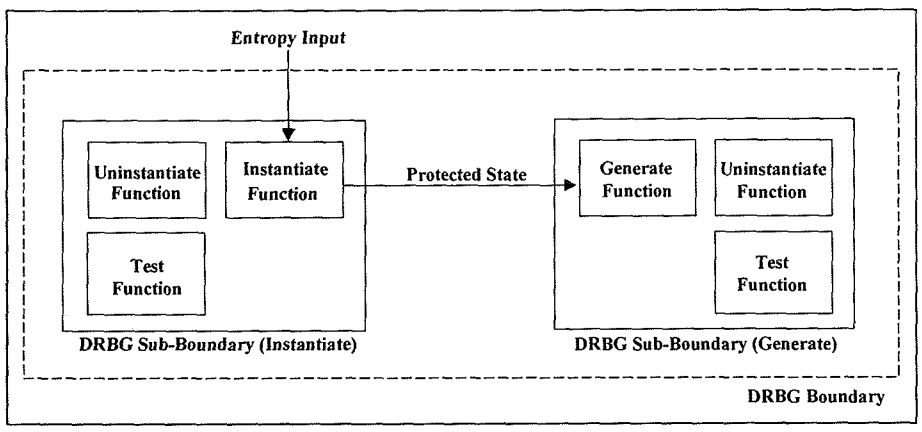

Figure 4: Distributed DRBG Functions

Each DRBG boundary or sub-boundary **shall** contain a test function to test the "health" of other DRBG functions within that boundary. In addition, each boundary or sub-boundary **shall** contain an uninstantiate function in order to perform and/or react to health testing.

When DRBG functions are distributed, appropriate mechanisms **shall** be used to protect the confidentiality and integrity of the internal state or parts of the internal state that are transferred between the distributed DRBG sub-boundaries. The confidentiality and integrity mechanisms and security strength **shall** be consistent with the data to be protected by the DRBG's consuming application (see SP 800-57).

8.6 Seeds

When a DRBG is used to generate pseudorandom bits, a seed **shall** be acquired prior to the generation of output bits by the DRBG. The seed is used to instantiate the DRBG and determine the initial internal state.

Reseeding is a means of ~~recovering~~ restoring the secrecy of the output of the DRBG if a seed or the internal state becomes known. Periodic reseeding is a good way of addressing the threat of either the DRBG seed, entropy input or working state being compromised over time. In some implementations (e.g., smartcards), an adequate reseeding process may not be possible. In these cases, the best policy might be to replace the DRBG, obtaining a new seed in the process (e.g., obtain a new smart card).

The seed and its use by a DRBG **shall** be generated and handled as specified in the following subsections.

8.6.1 Seed Construction for Instantiation

Figure 5 depicts the seed construction process for instantiation. The seed material used to determine a seed for instantiation consists of entropy input, a nonce and an optional personalization string. Entropy input **shall** always be used in the construction of a seed; requirements for the entropy input are discussed in Section 8.6.3. Except for the case noted below, Aa nonce **shall** also be used; requirements for the nonce are discussed in Section 8.6.7. This Recommendation also advises the inclusion of a personalization string; requirements for the personalization string are discussed in Section 8.7.12.

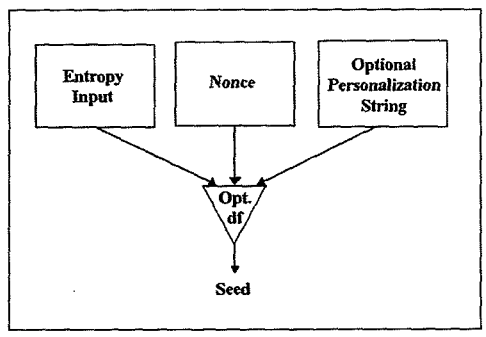

Figure 5: Seed Construction for Instantiation

Depending on the DRBG and the source of the entropy input, a derivation function may be required to derive a seed from the seed material. When full entropy input is readily available, the DRBG based on block cipher algorithms (see Section 10.2) may be implemented without a derivation function. When implemented in this manner, a nonce (as shown in Figure 5) is not used. Note, however, that the personalization string could contain a nonce, if desired.

8.6.2 Seed Construction for Reseeding

Figure 6 depicts the seed construction process for reseeding an instantiation. The seed material for reseeding consists of a value that is carried in the internal state[3], new entropy input and, optionally, additional input. The internal state value and the entropy input are required; requirements for the entropy input are discussed in Section 8.6.3. Requirements for the additional input are discussed in Section 8.7.23. As in Section 8.6.1, a derivation function may be required for reseeding. See Section 8.6.1 for further guidance.

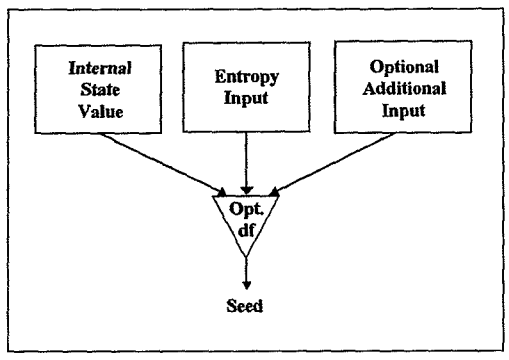

Figure 6: Seed Construction for Reseeding

[3] See each DRBG specification for the value that is used.

8.6.3. Entropy Requirements for the Entropy Input

The entropy input **shall** have entropy that is equal to or greater than the security strength of the instantiation. The entropy input for the seed **shall** contain sufficient entropy for the desired security strength. Additional entropy maymay be provided in the nonce or the optional personalization string during instantiation, or in the additional input during reseeding it generation, but this is not required. The use of more entropy than the minimum value will offer a security "cushion". This may be useful if the assessment of the entropy provided in the entropy input is incorrect. Having more entropy than the assessed amount is acceptable; having less entropy than the assessed amount could be fatal to security. The presence of more entropy than is required, especially during the instantiation, will provide a higher level of assurance than the minimum required entropy.

Entropy contained in the seed components is distributed across the seed (e.g., using an appropriate derivation function) by the instantiate and reseed functions.

The entropy input shall have entropy that is equal to or greater than the security strength of the instantiation. Note that the use of more entropy than the minimum value will offer a security "cushion". This may be useful if the assessment of the entropy provided in the entropy input is incorrect. Having more entropy than the assessed amount is acceptable; having less entropy than the assessed amount could be fatal to security. The presence of more entropy than is required, especially during the instantiation, will provide a higher level of assurance than the minimum required entropy.

8.6.4 Seed Length

The minimum length of the seed depends on the DRBG and the security strength required by the consuming application. See Section 10.

8.6.5 Entropy Input Source

The source of the entropy input **mayshall** be either:

1. aAn Approved NRBG,
2. aAn Approved DRBG, thus forming a chain of at least two DRBGs; the highest-level DRBG in the chain **shall** be seeded by an Approved NRBG or an entropy source, or
3. or anotherAn entropy source.

Further discussion about the entropy and entropy input sources is provided in Appendix C; discussion on RBG construction is provided in Appendix D.

8.6.6 Entropy Input and Seed Privacy

The entropy input and the resulting seed **shall** be handled in a manner that is consistent with the security required for the data protected by the consuming application. For example, if the DRBG is used to generate keys, then the entropy inputs and seeds used to generate the keys **shall** (at a minimum) be treated protected at least as well as the key (at a

~~minimum).~~

8.6.7 Nonce

A nonce ~~is~~ may be required in the construction of a seed during instantation in order to provide a security cushion to block certain attacks. The nonce **shall** be either:

a. A random value with at least (*security_strength*/2) bits of entropy,

b. A non-random value that is expected to repeat no more often than a (*security_strength*/2)-bit random string would be expected to repeat.

For case a, the nonce ~~may~~may be acquired from the same source and at the same time as the entropy input. In this case, the seed could be considered to be constructed from an "extra strong" entropy input and the optional personalization string, where the entropy for the entropy input is equal to or greater than (3/2 *security_strength*) bits.

The nonce is required for instantiation to provide *security_strength* bits of security. When a DRBG is instantiated many times without a nonce, a compromise may become more likely. In some consuming applications, a single DRBG compromise may reveal long-term secrets (e.g., a compromise of the DSA per-message secret reveals the signing key).

8.6.8 Reseeding

Generating too many outputs from a seed (and other input information) may provide sufficient information for successfully predicting future outputs unless prediction resistance is provided (see Section 8.8). Periodic reseeding will reduce security risks, reducing the likelihood of a compromise of the data that is protected by cryptographic mechanisms that use the DRBG.

Seeds **shall** have a finite seedlife (i.e., the length of the seed period); the maximum seedlife is dependent on the DRBG used. Reseeding is accomplished by 1) an explicit reseeding of the DRBG by the consuming application, or 2) by the generate function when prediction resistance is requested (see Section 8.8) or the limit of the seedlife is reached.

Reseeding of the DRBG **shall** be performed in accordance with the specification for the given DRBG. The DRBG reseed specifications within this Recommendation are designed to produce a new seed that is determined by both the old seed and newly-obtained entropy input that will support the desired security strength.

An alternative to reseeding is to create an entirely new instantiation. However, reseeding is preferred over creating a new instantiation. If there is an undetected failure in the entropy input source, a reseeded DRBG instantiation will still retain any previous entropy, whereas a re-instantiated DRBG may not have sufficient entropy to support the requested security strength.

8.6.9 Seed Use

A seed that is used to initialize one instantiation of a DRBG **shall not** be intentionally used to reseed the same instantiation or used as a seed for another DRBG instantiation. Note

that a DRBG does not provide output until a seed is available, and the internal state has been initialized (see Section 10).

8.6.10 Seed Separation

Seeds used by DRBGs and the entropy input used to create those seeds **shall not** be used for other purposes (e.g., domain parameter or prime number generation).

8.7 Other Inputs to the DRBG

Other input may be provided during DRBG instantiation, pseudorandom bit generation and reseeding. This input may contain entropy, but this is not required. During instantiation, a personalization string may be provided and combined with entropy input and a nonce to derive a seed (see Section 8.6.1). When pseudorandom bits are requested and when reseeding is performed, additional input may be provided.

Depending on the method for acquiring the input, the exact value of the input may or may not be known to the user or consuming application. For example, the input could be derived directly from values entered by the user or consuming application, or the input could be derived from information introduced by the user or consuming application (e.g., from timing statistics based on key strokes), or the input could be the output of another DRBG or an NRBG.

8.7.1 Personalization String

During instantiation, a personalization string **should** be used to derive the seed (see Section 8.6.1). The intent of a personalization string is to differentiate this DRBG instantiation from all others that might ever be created. The personalization string **should** be set to some bitstring that is as unique as possible, and ~~may~~may include secret information. The value of any secret information contained in the personalization string **should** be no greater than the claimed strength of the DRBG, as the DRBG's cryptographic mechanisms will protect this information from disclosure. Good choices for the personalization string contents include:

- Device serial numbers,
- Public keys,
- User identification,
- Private keys,
- PINs and passwords,
- Secret per-module or per-device values,
- Timestamps,
- Network addresses,
- Special secret key values for this specific DRBG instantiation,
- Application identifiers,
- Protocol version identifiers,
- Random numbers, and
- Nonces.

8.7.2 Additional Input

During each request for bits from a DRBG and during reseeding, the insertion of additional input is allowed. This input is optional, and the ability to enter additional input may or may not be included in an implementation. Additional input may be either secret or publicly known; its value is arbitrary, although its length may be restricted, depending on the implementation and the DRBG. The use of additional input may be a means of providing more entropy for the DRBG internal state that will increase assurance that the entropy requirements are met. If the additional input is kept secret and has sufficient entropy, the input can provide more assurance when recovering from the compromise of the seed or one or more DRBG internal states.

8.8 Prediction Resistance and Backtracking Resistance

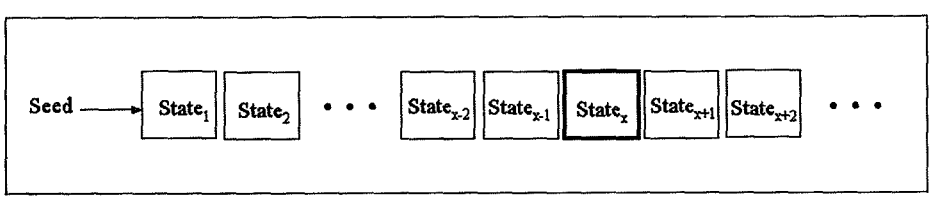

Figure 7: Sequence of DRBG States

Figure 7 depicts the sequence of DRBG internal states that result from a given seed. Some subset of bits from each internal state are used to generate pseudorandom bits upon request by a user. The following discussions will use the figure to explain backtracking and prediction resistance.

Suppose that a compromise occurs at $State_x$, where $State_x$ contains both secret and public information.

Backtracking Resistance: Backtracking resistance means that a compromise of the DRBG internal state has no effect on the security of prior outputs. That is, an adversary who is given access to all or any subset of that prior output sequence cannot distinguish it from random; if the adversary knows only part of the prior output, he cannot determine any bit of that prior output sequence that the adversary he has not already seen. In other words, a compromise has no effect on the security of prior outputs.

For example, suppose that an adversary knows *State* ... and also knows the output bits from *State*$_1$ to *State*$_{x-1}$. Backtracking resistance means that:

a. The output bits from *State*$_1$ to *State*$_{x-1}$ cannot be distinguished from random;

b. The prior internal state values themselves (*State*$_1$ to *State*$_{x-1}$) cannot be recovered, given knowledge of the secret information in *State*$_x$. Some or all of the output bits cannot be determined from knowledge of *State*$_x$, that is, cannot be

Backtracking resistance can be provided by ensuring that the DRBG algorithm is a one-way function. All DRBGs in this Recommendation have been designed to provide backtracking resistance.

Prediction Resistance: Prediction resistance means that a compromise of the DRBG internal state has no effect on the security of future DRBG outputs. That is, an adversary who is given access to all of the output sequence after the compromise cannot distinguish it from random; if the adversary knows only part of the future output sequence, he cannot predict any bit of that future output sequence that he has not already seen.

For example, suppose that an adversary knows $State_x$. Prediction resistance means that:

a. The output bits from $State_x$ and forward cannot be distinguished from an ideal random bitstring by the adversary.

b. The future internal state values themselves ($State_{x+1}$ and forward) cannot be predicted, given knowledge of $State_x$.

Prediction resistance can be provided only by ensuring that a DRBG is effectively reseeded between DRBG requests. That is, an amount of entropy that is sufficient to support the security strength of the DRBG (i.e., an amount that is at least equal to the security strength) must be provided to the DRBG in a way that ensures that knowledge of the current internal DRBG internal state does not allow an adversary any useful knowledge about future DRBG internal states or outputs. Prediction resistance is provided in this Recommendation by the use of a prediction resistance flag.

9 DRBG Functions

The DRBG functions in this Recommendation are specified as an algorithm and an "envelope" of pseudocode around that algorithm. The pseudocode in the envelopes (provided in this section) checks the input parameters, obtains input not provided by the input parameters, accesses the appropriate DRBG algorithm and handles the internal state. A function need not be implemented using such envelopes, but the function **shall** have equivalent functionality.

In the specifications of this Recommendation, a **Get_entropy_input** pseudo-function is used for convenience. This function is not fully specified in this Recommendation, but has the following meaning:

Get_entropy_input: A function that is used to obtain entropy input. The function call is:

(*status*, *entropy_input*) = **Get_entropy_input** (*min_entropy*, *min_ length*, *max_ length*)

which requests a string of bits (*entropy_input*) with at least *min_entropy* bits of entropy. The length for the string **shall** be equal to or greater than *min_length* bits, and less than or equal to *max_length* bits. A *status* code is also returned from the function.

Note that an implementation may choose to define this functionality differently; for example, for many of the DRBGs, the *min_length* = *min_entropy* for the **Get_entropy_input** function, in which case, the second parameter could be omitted.

9.1 Instantiating a DRBG

A DRBG **shall** be instantiated prior to the generation of pseudorandom bits. The instantiate function:

1. Checks the validity of the input parameters,
2. Determines the security strength for the DRBG instantiation,
3. Determines any DRBG specific parameters (e.g., elliptic curve domain parameters),
4. Obtains entropy input with entropy sufficient to support the security strength,
5. Obtains the nonce (if required),
6. Determines the initial internal state using the instantiate algorithm,
7. Returns a *state_handle* for the internal state to the consuming application (see below).

Let *working_state* be the working state for the particular DRBG, and let *min_length*, *max_ length*, and *highest_supported_security_strength* be defined for each DRBG (see Section 10).

The following or an equivalent process **shall** be used to instantiate a DRBG.

Input from a consuming application for instantiation:

1. *requested_instantiation_security_strength*: A requested security strength for the instantiation. DRBG implementations that support only one security strength do not require this parameter; however, any consuming application using that DRBG implementation must be aware of this limitation.
2. *prediction_resistance_flag*: Indicates whether or not prediction resistance may be required by the consuming application during one or more requests for pseudorandom bits. DRBGs that are implemented to always or never support prediction resistance do not require this parameter. However, the user of a consuming application must determine whether or not prediction resistance may be required by the consuming application before electing to use such a DRBG implementation. If the *prediction_resistance_flag* is not needed (i.e., because prediction resistance is always or never performed), then the input parameter may be omitted, and the *prediction_resistance_flag* may be omitted from the internal state in step 11 of the instantiate process.
3. *personalization_string*: An optional input that provides personalization information (see Sections 8.6.1 and 8.7.1). The maximum length of the personalization string (*max_personalization_string_length*) is implementation dependent, but **shall** be less than or equal to the maximum length specified for the given DRBG (see Section 10). If a personalization string will never be used, then the input parameter and step 3 of the instantiate process may be omitted, and process step 9 may be modified to omit the personalization string.

Required information not provided by the consuming application during instantiation:

Comment: This input **shall not** be provided by the consuming application as an input parameter during the instantiate request.

1. *entropy_input*: Input bits containing entropy. The maximum length of the *entropy_input* is implementation dependent, but **shall** be less than or equal to the specified maximum length for the selected DRBG (see Section 10).
2. *nonce*: A nonce as specified in Section 8.6.7. Note that if a random value is used as the nonce, the *entropy_input* and *nonce* could be acquired using a single **Get_entropy_input** call (see step 6 of the instantiate process); in this case, the first parameter would be adjusted to include the entropy for the *nonce* (i.e., *security_strength* would be increased by at least *security_strength*/2), process step 8 would be omitted, and the *nonce* would be omitted from the parameter list in process step 9.

Output to a consuming application after instantiation:

1. *status*: The status returned from the instantiate function. The *status* will indicate **SUCCESS** or an **ERROR**. If an **ERROR** is indicated, either no *state_handle* or an invalid *state_handle* **shall** be returned. A consuming application **should** check the *status* to determine that the DRBG has been correctly instantiated.
2. *state_handle*: Used to identify the internal state for this instantiation in subsequent calls to the generate, reseed, uninstantiate and test functions.

Information retained within the DRBG boundary after instantiation:

The internal state for the DRBG, including the *working_state* and administrative information (see Sections 8.3 and 10).

Instantiate Process:

Comment: Check the validity of the input parameters.

1. If *requested_instantiation_security_strength* > *highest_supported_security_strength*, then returuen an **ERROR_FLAG**.
2. If *prediction_resistance_flag* is set, and prediction resistance is not supported, then return an **ERROR_FLAG**.
3. If the length of the *personalization_string* > *max_personalization_string_length*, return an **ERROR_FLAG**.
4. Set *security_strength* to the nearest security strength greater than or equal to *requested_instantiation_security_strength*.

   Comment: The following step is required by the Dual_EC_DRBG when multiple curves are available (see Section 10.3.2.21.2). Otherwise, the step **should** be omitted.
5. Using *security_strength*, select appropriate DRBG parameters.

   Comment: Obtain the entropy input.
6. (*status*, *entropy_input*) = **Get_entropy_input** (*security_strength*, *min_length*, *max_length*).
7. If an **ERROR** is returned in step 6, return an **CATASTROPHIC_ERROR_FLAG**.
8. Obtain a *nonce*.

   Comment: This step **shall** include any appropriate checks on the acceptability of the *nonce*. See Section 8.6.7.

   Comment: Call the appropriate instantiate algorithm in Section 10 to obtain values for the initial *working_state*.

9. *initial_working_state* = **Instantiate_algorithm** (*entropy_input*, *nonce*, *personalization_string*).
10. Get a *state_handle* for a currently empty internal state. If an unused internal state cannot be found, return an **ERROR_FLAG**.
11. Set the internal state indicated by *state_handle* to the initial values for the internal state (i.e., set the *working_state* to the values returned as *initial_working_state* in step 9 and any other values required for the *working_state* (see Section 10), and set the ~~*working_state* and~~ administrative information to the appropriate values (e.g., the values of *security_strength* and the *prediction_resistance_flag*). ~~as appropriate.~~
12. Return **SUCCESS** and *state_handle*.

9.2 Reseeding a DRBG Instantiation

The reseeding of an instantiation is not required, but is recommended whenever a comsuming application and implementation are able to perform this process. Reseeding will insert additional entropy into the generation of pseudorandom bits. Reseeding may be:

- explicitly requested by a consuming application,
- performed when prediction resistance is requested by a consuming application,
- triggered by the generate function when a predetermined number of pseudorandom outputs have been produced or a predetermined number of generate requests have been made (i.e., at the end of the seedlife), or
- triggered by external events (e.g., whenever sufficient entropy is available).

If a reseed capability is not available, a new DRBG instantiation may be created (see Section 9.1).

The reseed function:

1. Checks the validity of the input parameters,
2. Obtains entropy input with sufficient entropy to support the security strength, and
3. Using the reseed algorithm, combines the current working state with the new entropy input and any additional input to determine the new working state.

Let *working_state* be the working state for the particular DRBG, and let *min_length* and *max_ length* be defined for each DRBG (see Section 10).

The following or an equivalent process **shall** be used to reseed the DRBG instantiation.

Input from a consuming application for reseeding:

1) *state_handle*: A pointer or index that indicates the internal state to be reseeded. This value was returned from the instantiate function specified in Section 9.1.
2) *additional_input*: An optional input. The maximum length of the *additional_input*

(*max_additional_input_length*) is implementation dependent, but **shall** be less than or equal to the maximum value specified for the given DRBG (see Section 10). If *additional_input* will never be used, then the input parameter and step 2 of the reseed process may be omitted, and step ~~2~~ 5 may be modified to remove the *additional_input* from the parameter list.

Required information not provided by the consuming application during reseeding:

Comment: This input **shall not** be provided by the consuming application in the input parameters.

1. *entropy_input*: Input bits containing entropy. The maximum length of the *entropy_input* is implementation dependent, but **shall** be less than or equal to the specified maximum length for the selected DRBG (see Section 10).
2. Internal state values required by the DRBG for reseeding for the *working_state* and administrative information, as appropriate.

Output to a consuming application after reseeding:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS** or an **ERROR**.

Information retained within the DRBG boundary after reseeding:

Replaced internal state values (i.e., the *working_state*).

Reseed Process:

Comment: Get the current internal state and check the input parameters.

1. Using *state_handle*, obtain the current internal state. If *state_handle* indicates an invalid or unused internal state, return an **ERROR_FLAG**.
2. If the length of the *additional_input* > *max_additional_input_length*, return an **ERROR_FLAG**.

Comment: Obtain the entropy input.

3. (*status*, *entropy_input*) = **Get_entropy_input** (*security_strength*, *min_length*, *max_length*).
4. If an **ERROR** is returned in step 3, return a **CATASTROPHIC ~~R~~ ERROR_FLAG**.

Comment: Get the new *working_state* using the appropriate reseed algorithm in Section 10.

5. *new_working_state* = **Reseed_algorithm** (*working_state*, *entropy_input*, *additional_input*).

6. Replace the *working_state* in the internal state indicated by *state_handle* with the new values of *new_working_state* obtained in step 5.
7. Return **SUCCESS**.

9.3 Generating Pseudorandom Bits Using a DRBG

This function is used to generate pseudorandom bits after instantiation or reseeding. The generate function:

1. Checks the validity of the input parameters.
2. Calls the reseed function to obtain sufficient entropy if the instantiation needs additional entropy because the end of the seedlife has been reached or prediction resistance is required; see Sections 9.3.2 and 9.3.3 for more information on reseeding at the end of the seedlife and on handling prediction resistance requests.
3. Generates the requested pseudorandom bits using the generate algorithm.
4. Updates the working state.
5. Returns the requested pseudorandom bits to the consuming appication.

9.3.1 The Generate Function

Let *outlen* be the length of the output block of the cryptographic primitive (see Section 10).

The following or an equivalent process **shall** be used to generate pseudorandom bits.

Input from a consuming application for generation:

1. *state_handle*: A pointer or index that indicates the internal state to be used.
2. *requested_number_of_bits*: The number of pseudorandom bits to be returned from the generate function. The *max_number_of_bits_per_request* is implementation dependent, but **shall** be less than or equal to the value provided in Section 10 for a specific DRBG.
3. *requested_security_strength*: The security strength to be associated with the requested pseudorandom bits. DRBG implementations that support only one security strength do not require this parameter; however, any consuming application using that DRBG implementation must be aware of this limitation.
4. *prediction_resistance_request*: Indicates whether or not prediction resistance is to be provided. DRBGs that are implemented to always or never support prediction resistance do not require this parameter. However, the user of a consuming application must determine whether or not prediction resistance may be required by the application before electing to use such a DRBG implementation.

   If prediction resistance is never provided, then the *prediction_resistance_request* input parameter and step 5 of the generate process may be omitted, and step 7 may be modified to omit the check for the *prediction_resistance_request.*

If prediction resistance is always performed, then the *prediction_resistance_request* input parameter and step 5 may be omitted, and steps 7 and 8 are replaced by:

status = **Reseed** (*state_handle*, *additional_input*).

If *status* indicates an **ERROR**, then return *status*.

Using *state_handle*, obtain the new internal state.

(*status*, *pseudorandom_bits*, *working_state*) = **Generate_algorithm** (*working_state*, *requested_number_of_bits*).

Note that if *additional_input* is never provided, then the *additional_input* parameter in the Reseed call above may be omitted.

5. *additional_input*: An optional input. The maximum length of the *additional_input* (*max_additional_input_length*) is implementation dependent, but **shall** be less than or equal to the specified maximum length for the selected DRBG (see Section 10). If *additional_input* will never be used, then the input parameter, process step 4, step 7.4 and the *additional_input* input parameter in step 8 may be omitted.

Required information not provided by the consuming application during generation:

1. Internal state values required for generation for the *working_state* and administrative information, as appropriate.

Output to a consuming application after generation:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS** or an **ERROR**.
2. *pseudorandom_bits*: The pseudorandom bits that were requested.

Information retained within the DRBG boundary after generation:

Replaced internal state values (i.e., the *working_state*).

Generate Process:

Comment: Get the internal state and check the input parameters.

1. Using *state_handle*, obtain the current internal state for the instantiation. If *state_handle* indicates an invalid or unused internal state, then return an **ERROR_FLAG**.
2. If *requested_number_of_bits* > *max_number_of_bits_per_request*, then return an **ERROR_FLAG**.
3. If *requested_security_strength* > the *security_strength* indicated in the internal state, then return an **ERROR_FLAG**.
4. If the length of the *additional_input* > *max_additional_input_length*, then return an **ERROR_FLAG**.

5. If *prediction_resistance_request* is set, and *prediction_resistance_flag* is not set, then return an **ERROR_FLAG**.

6. Clear the *reseed_required_flag*.

~~Comment: Get the requested pseudorandom bits.~~

7. If *reseed_required_flag* is set, or if *prediction_resistance_request* is set, then

Comment: Reseed the instantiation (see Section 9.2).

7.1 *status* = **Reseed** (*state_handle*, *additional_input*).

7.2 If *status* indicates an **ERROR**, then return ~~*an ERROR*~~*status*.

7.3 Using *state_handle*, obtain the new internal state.

7.4 *additional_input* = the *Null* string.

7.5 Clear the *reseed_required_flag*.

Comment: Request the generation of *pseudorandom_bits* using the appropriate generate algorithm in Section 10.

8. (*status*, *pseudorandom_bits*, *new_working_state*) = **Generate_algorithm** (*working_state*, *requested_number_of_bits*, *additional_input*).

9. If *status* indicates that a reseed is required before the requested bits can be generated, then

9.1 Set the *reseed_required_flag*.

9.2 Go to step 7.

10. ~~If an **ERROR** is returned from step 8,~~

~~10.1 Delete all instantiations using the uninstantiate function.~~

~~10.2 Return the **ERROR** received from step 8.~~

~~11.~~ Replace the old *working_state* in the internal state indicated by *state_handle* with the ~~new~~ values of *new_working_state*.

~~12~~11. Return **SUCCESS** and *pseudorandom_bits*.

Implementation notes:

If a reseed capability is not available, then steps 6 and 7 may be removed; and step 9 is replaced by:

9. If *status* indicates that a reseed is required before the requested bits can be

generated, then

9.1 *status* = **Uninstantiate** (*state_handle*).

9.2 If an **ERROR** is returned in step 9.1, then return the **ERROR**.

9.3 Return an indication that the DRBG instantiation can no longer be used.

9.3.2 Reseeding at the End of the Seedlife

When pseudorandom bits are requested by a consuming application, the generate function checks whether or not a reseed is required by comparing the counter within the internal state (see Section 8.3) against a predetermined reseed interval for the DRBG implementation. This is specified in the generate function (see Section 9.3.1) as follows:

a. Step 6 clears the *reseed_required_flag*.
b. Step 7 checks the value of the *reseed_required_flag*. At this time, it is clear, so step 7 would be skipped unless prediction resistance was requested by the consuming application. For the purposes of this explanation, assume that prediction resistance was not requested.
c. Step 8 calls the **Generate_algorithm**, which will check whether a reseed is required. If it is required, an appropriate *status* will be returned.
d. Step 9 checks the *status* returned by the **Generate_algorithm**. If the *status* does not indicate that a reseed is required, the generate process continues with step 10.
e. If the status indicates that a reseed is required, then the *reseed_required_flag* is set, and processing continues by going back to step 7.
f. The substeps in step 7 are executed. The reseed function will be called; any *additional_input* provided by the consuming application in the generate request will be used during reseeding. Then the new values of the internal state are acquired, any *additional_input* provided by the consuming application in the generate request is replaced by a *Null* string, and the *reseed_required_flag* is cleared.
g. The generate algorithm is called (again) in step 8, the check of the returned *status* is made in step 9, and (presumably) step 10 is then executed.

9.3.3 Handling Prediction Resistance Requests

When pseudorandom bits are requested by a consuming application with prediction resistance, the generate function specified in Section 9.3.1 checks that the instantiation allows prediction resistance requests (see step 5 of the generate process); clears the *reseed_required_flag* (even though the flag won't be used in this case); executes the substeps of step 7, resulting in a reseed and a new internal state for the instantiation; obtains pseudorandom bits (see step 8); passes through step 9, since another reseed will not be required; and continues with step 10.

9.4 Removing a DRBG Instantiation

The internal state for an instantiation may need to be "released" by erasing the contents of the internal state. The uninstantiate function:

1. Checks the input parameter for validity.
2. Empties the internal state.

The following or an equivalent process **shall** be used to remove (i.e., uninstantiate) a DRBG instantiation:

Input from a consuming application for uninstantiation:

1. *state_handle*: A pointer or index that indicates the internal state to be "released".

Output to a consuming application after uninstantiation:

1. *status*: The status returned from the function. The status will indicate **SUCCESS** or **ERROR_FLAG**.

Information retained within the DRBG boundary after uninstantiation:

An empty internal state.

Uninstantiate Process:

1. If *state_handle* indicates an invalid state, then return an **ERROR_FLAG**.
2. Erase the contents of the internal state indicated by *state_handle*.
3. Return **SUCCESS**.

~~9.5 Auxilliary Functions~~

~~Derivation functions are internal functions that are used during DRBG instantiation and reseeding to either derive internal state values or to distribute entropy throughout a bitstring. Two methods are provided. One method is based on hash functions (see Section 9.6.1), and the other method is based on block cipher algorithms (see 9.6.2). The block cipher derivation function uses a **Block_Cipher_Hash** function that is specified in Section 9.6.3.~~

~~**9.5.1 Derivation Function Using a Hash Function (Hash_df)**~~

~~This derivation function is used by the **Hash_DRBG** and **Dual_EC_DRBG** specified in Section 10. The hash-based derivation function hashes an input string and returns the requested number of bits. Let **Hash (...)** be the hash function used by the DRBG, and let *outlen* be its output length.~~

~~The following or an equivalent process **shall** be used to derive the requested number of bits.~~

Input:

1. *input_string*: The string to be hashed.
2. *no_of_bits_to_return*: The number of bits to be returned by **Hash_df.** The maximum length (*max_number_of_bits*) is implementation dependent, but **shall** be less than or equal to (*255 × outlen*). *no_of_bits_to_return* is represented as a 32-bit integer.

Output:

1. *status*: The status returned from **Hash_df.** The status will indicate **SUCCESS** or **ERROR.**
2. *requested_bits* : The result of performing the **Hash_df.**

Process:

1. If *no_of_bits_to_return* > *max_number_of_bits*, then return an **ERROR.**
2. *temp* = the Null string.
3. $len = \left\lceil \frac{no\_of\_bits\_to\_return}{outlen} \right\rceil$.
4. *counter* = an 8-bit binary value representing the integer "1".
5. For $i = 1$ to *len* do

   Comment : In step 5.1, *no_of_bits_to_return* is used as a 32-bit string.

   5.1 *temp* = *temp* || **Hash** (*counter* || *no_of_bits_to_return* || *input_string*).

   5.2 *counter* = *counter* + 1.
6. *requested_bits* = Leftmost (*no_of_bits_to_return*) of *temp*.
7. Return **SUCCESS** and *requested_bits*.

9.5.2 Derivation Function Using a Block Cipher Algorithm (Block_Cipher_df)

This derivation function is used by the **CTR_DRBG** that is specified in Section 10.2. Let **Block_Cipher_Hash** be the function specified in Section 9.6.3. Let Let *outlen* be its output block length, which is a multiple of 8 bits for the Approved block cipher algorithms. and let *keylen* be the key length.

The following or an equivalent process **shall** be used to derive the requested number of bits.

Input:

1. *input_string*: The string to be operated on. This string **shall** be a multiple of 8 bits.
2. *no_of_bits_to_return*: The number of bits to be returned by **Block_Cipher_df.** The

maximum length (*max_number_of_bits*) is 512 bits for the currently approved block cipher algorithms.

Output:

1. *status*: The status returned from **Block_Cipher_df**. The status will indicate **SUCCESS** or **ERROR**.
2. *requested_bits* : The result of performing the **Block_Cipher_df**.

Process:

1. If (*number_of_bits_to_return* > *max_number_of_bits*), then return an **ERROR**.
2. *L* = **len** (*input_string*)/8. Comment: *L* is the bitstring represention of the integer resulting from **len** (*input_string*)/8. *L* **shall** be represented as a 32-bit integer.
3. *N* = *number_of_bits_to_return*/8. Comment : *N* is the bitsting represention of the integer resulting from *number_of_bits_to_return*/8. *N* **shall** be represented as a 32-bit integer.

   Comment: Prepend the string length and the requested length of the output to the *input_string*.
3. *S* = *L* || *N* || *input_string* || 0x80.

   Comment : Pad *S* with zeros, if necessary.
4. While (**len** (*S*) mod *outlen*) ≠ 0, *S* = *S* || 0x00.

   Comment : Compute the starting value.
5. *temp* = the *Null* string.
6. *i* = 0. Comment : *i* **shall** be represented as a 32-bit integer, i.e., **len** (*i*) = 32.
7. *K* = Leftmost *keylen* bits of 0x010203...1F.
8. While **len** (*temp*) < *keylen* + *outlen*, do
   8.1 *IV* = *i* || $0^{outlen - \text{len}(i)}$. Comment: The 32-bit integer represenation of *i* is padded with zeros to *outlen* bits.
   8.2 *temp* = *temp* || **Block_Cipher_Hash** (*K*, (*IV* || *S*)).
   8.3 *i* = *i* + 1.

   Comment: Compute the requested number of bits.

9. *K* = Leftmost *keylen* bits of *temp*.

10. *X* = Next *outlen* bits of *temp*.

11. *temp* = the *Null* string.

12. While **len** (*temp*) < *number_of_bits_to_return*, do

    12.1 *X* = **Block_Encrypt** (*K*, *X*).

    12.2 *temp* = *temp* || *X*.

13. *requested_bits* = Leftmost *number_of_bits_to_return* of *temp*.

14. Return **SUCCESS** and *requested_bits*.

9.5.4 Block_Cipher_Hash Function

Let *outlen* be the length of the output block of the block cipher algorithm to be used.

The following or an equivalent process **shall** be used to derive the requested number of bits.

Input:

1. *Key*: The key to be used for the block cipher operation.

2. *data_to_hash*: The data to be operated upon. Note that the length of *data_to_hash* must be a multiple of *outlen*. This is guaranteed by steps 4 and 8.1 in Section 9.6.2.

Output:

1. *output_block*: The result to be returned from the **Block_Cipher_Hash** operation.

Process:

1. *chaining_value* = 0^{outlen}. Comment: Set the first chaining value to *outlen* zeros.

2. *n* = **len** (*data_to_hash*)/*outlen*.

3. Split the *data_to_hash* into *n* blocks of *outlen* bits each forming $block_1$ to $block_n$.

4. For *i* = 1 to *n* do

    4.1 *input_block* = *chaining_value* $\oplus$ $block_i$.

    4.2 *chaining_value* = **Block_Encrypt** (*Key*, *input_block*).

5. *output_block* = chaining_value.

6. Return *output_block*.

9.65 Self-Testing of the DRBG

A DRBG **shall** perform self testing to obtain assurance that the implementation continues to operate as designed and implemented (health testing). The testing function(s) within a DRBG boundary (or sub-boundary) **shall** test each DRBG function within that boundary.

Note that this may require the creation and use of an instantiation for testing purposes only.

Errors occurring during testing **shall** be perceived as complete DRBG failures. The condition causing the failure **shall** be corrected and the DRBG re-instantiated before requesting pseudorandom bits (also, see Section 9.76)

9.65.1 Testing the Instantiate Function

Known-answer tests on the instantiate function **shall** be performed prior to creating each operational instantiation. However, if several instantiations are performed in quick succession using the same input parameters, then the testing maymay be reduced to testing only prior to creating the first instantiation using that parameter set until such time as the succession of instantiations is completed. Thereafter, other instantiations **shall** be tested as specified above.

The *security_strength* and *prediction_resistance_flag* to be used in the operational invocation **shall** be used during the test. Representative fixed values and lengths of the *entropy_input*, *nonce* and *personalization_string* (if allowed) **shall** be used; the value of the *entropy_input* used during testing **shall not** be intentionally reused during normal operations (either by the instantiate or the reseed functions). Error handling **shall** also be tested, including whether or not the instantiate function handles an error from the entropy input source correctly.

If the values used during the test produce the expected results, and errors are handled correctly, then the instantiate function may be used to instantiate using the tested values of *security_strength* and *prediction_resistance_flag*.

An implementation **should** provide a capability to test the instantiate function on demand.

9.65.2 Testing the Generate Function

Known-answer tests **shall** be performed on the generate function before the first use of the function and at reasonable intervals defined by the implementer. The implementer **shall** document the intervals and provide a justification for the selected intervals.

The known-answer tests **shall** be performed for each implemented *security_strength*. Representative fixed values and lengths for the *requested_number_of_bits* and *additional_input* (if allowed) and the working state of the internal state value (see Sections 8.3 and 10) **shall** be used. If prediction resistance is available, then each combination of the *security_strength*, *prediction_resistance_request* and *prediction_resistance_flag* **shall** be tested. The error handling for each input parameter **shall** also be tested, and testing **shall** include setting the *reseed_counter* to meet or exceed the *reseed_interval* in order to check that the implementation is reseeded or that the DRBG is "shut down", as appropriate.

If the values used during the test produce the expected results, and errors are handled correctly, then the generate function may be used during normal operations.

Bits generated during health testing **shall not** be output as pseudorandom bits.

An implementation **should** provide a capability to test the generate function on demand.

~~Note that the generate function performs a continuous test by comparing sequential output blocks.~~

9.65.3 Testing the Reseed Function

A known-answer test of the reseed function **shall** use the *security_strength* in the internal state of the instantiation to be reseeded. Representative values of the *entropy_input* and *additional_input* (if allowed) and the working state of the internal state value **shall** be used (see Sections 8.3 and 10). Error handling **shall** also be tested, including an error in obtaining the *entropy_input* (e.g., the *entropy_input* source is broken).

If the values used during the test produce the expected results, and errors are handled correctly, then the reseed function may be used to reseed the instantiation.

The reseed function may be called every time that the generate function is called if prediction resistance is available, and considerbly less frequently otherwise. Self-testing **shall** be performed as follows:

1. When prediction resistance is available in an implementation, the reseed function **shall** be tested whenever the generate function is tested (see above).
2. When prediction resistance is not available in an implementation, the reseed function **shall** be tested whenever the reseed function is invoked and before the reseed is performed on the operational instantiation.

An implementation **should** provide a capability to test the reseed function on demand.

9.65.4 Testing the Uninstantiate Function

The uninstantiate function **shall** be tested whenever other functions are tested. Testing **shall** attempt to demonstrate that error handling is performed correctly, and the internal state has been zeroized.

9.76 Error Handling

The expected errors are indicated for each DRBG function (see Sections 9.1 - 9.4) and for the derivation functions in Section ~~9.5~~10.4. The error handling routines **should** indicate the type of error.

9.6.1 Errors Encountered During Normal Operation

Many errors during normal operation may be caused by a consuming application's improper DRBG request; these errors are indicated by "**ERROR_FLAG**" in the pseudocode.~~.~~ In these cases, the consuming application user is responsible for correcting the request within the limits of the user's organizational security policy. For example, if a failure indicating an invalid requested security strength is returned, a security strength higher than the DRBG or the DRBG instantiation can support has been requested. The user ~~may~~may reduce the requested security strength if the organization's security policy allows

the information to be protected using a lower security strength, or the user **shall** use an appropriately instantiated DRBG.

For catastrophic errors (e.g., entropy input source failurei.e., those errors indicated by the **CATASTROPHIC_ERROR_FLAG** in the pseudocode), the DRBG **shall not** produce further output until the source of the error is corrected, and the DRBG is re-instantiated.

9.6.2 Errors Encountered During Self-Testing

During self-testing, all unexpected behavior is catastrophic. The DRBG **shall** be corrected, and the DRBG **shall** be re-instantiated before the DRBG can be used to produce pseudorandom bits. Examples of unexpected behavior include:

- A test deliberately inserts an error, and the error is not detected, or
- An incorrect result is returned from the instantiate, reseed or generate function than was expected.

10 DRBG Algorithm Specifications

Several DRBGs are specified in this Recommendation. The selection of a DRBG depends on several factors, including the security strength to be supported and what cryptographic primitives are available. An analysis of the consuming application's requirements for random numbers **shall** be conducted in order to select an appropriate DRBG. A detailed discussion on DRBG selection is provided in Appendix G. Pseudocode examples for each DRBG are provided in Appendix F. Conversion specifications required for the DRBG implementations (e.g., between integers and bitstrings) are provided in Appendix B.

10.1 Deterministic RBGs Based on Hash Functions

A DRBG may be based on a hash function that is non-invertible or one-way. The hash-based DRBGs specified in this Recommendation have been designed to use any Approved hash function and may be used by consuming applications requiring various security strengths, providing that the appropriate hash function is used and sufficient entropy is obtained for the seed.

The following are provided as DRBGs based on hash functions:

1. The **Hash_DRBG** specified in Section 10.1.1.
2. The **HMAC_DRBG** specified in Section 10.1.2.

The maximum security strength that can be supported by each hash function is provided in SP 800-57. However, this Recommendation supports only four security strengths: 112, 128, 192, and 256. Table 2 specifies the values that **shall** be used for the function envelopes and DRBG algorithm for each Approved hash function.

Table 2: Definitions for Hash-Based DRBGs

	SHA-1	**SHA-224**	**SHA-256**	**SHA-384**	**SHA-512**
Supported security strengths	See SP 800-57				
highest_supported_security_strength	See SP 800-57				
Output Block Length (*outlen*)	160	224	256	384	512
Required minimum entropy for instantiate and reseed	*security_strength*				
Minimum entropy input length (*min_length*)	*security_strength*				
Maximum entropy input length (*max_ length*)	$\leq 2^{35}$ bits				
Seed length (*seedlen*) for Hash_DRBG	440	440	440	888	888

	SHA-1	SHA-224	SHA-256	SHA-384	SHA-512
Maximum personalization string length (*max_personalization_string_length*)	$\leq 2^{35}$ bits				
Maximum additional_input length (*max_additional_input_length*)	$\leq 2^{35}$ bits				
max_number_of_bits_per_request	$\leq 2^{19}$ bits				
Number of requests between reseeds (*reseed_interval*)	$\leq 2^{48}$				

Note that since SHA-224 is based on SHA-256, there is no efficiency benefit for using the SHA-224; this is also the case for SHA-384 and SHA-512, i.e., the use of SHA-256 or SHA-512 instead of SHA-224 or SHA-384, respectively, is preferred.

The value for *seedlen* for Hash_DRBG is determined by subtracting the count field (in the hash function specification) and one byte of padding from the hash function input block length; in the case of SHA-1, SHA-224 and SHA 256, *seedlen* = 512 - 64 - 8 = 440; for SHA-384 and SHA-512, *seedlen* = 1024 - 128 - 8 = 888.

10.1.1 Hash_DRBG

Figure 8 presents the normal operation of the **Hash_DRBG** generate algorithm. The **Hash_DRBG** requires the use of a hash function during the instantiate, reseed and generate functions; the same hash function **shall** be used in all functions. **Hash_DRBG** uses the derivation function specified in Section 10.4.1 during instantiation and reseeding. The hash function to be used **shall** meet or exceed the desired security strength of the consuming application.

10.1.1.1 Hash_DRBG Internal State

The *internal_state* for **Hash_DRBG** consists of:

1. The *working_state*:
   a. A value (*V*) of *seedlen* bits that is updated during each call to the DRBG.
   b. A constant *C* of *seedlen* bits that depends on the *seed.*
   c. ~~The *previous_output_block*; this will be used to perform a continuous test on the output from the generate function.~~
   ~~d.~~ A counter (*reseed_counter*) that indicates the number of requests for pseudorandom bits since new *entropy_input* was obtained during instantiation or reseeding.
2. Administrative information:

a. The *security_strength* of the DRBG instantiation.

b. A *prediction_resistance_flag* that indicates whether or not a prediction resistance capability is required for the DRBG.

The values of *V* and *C* are the critical values of the internal state upon which the security of this DRBG depends (i.e., *V* and *C* are the "secret values" of the internal state).

10.1.1.2 Instantiation of Hash_DRBG

Notes for the instantiate function specified in Section 9.1:

The instantiation of **Hash_DRBG** requires a call to the instantiate function. Process step 9 of that function calls the instantiate algorithm in this section. For this DRBG, step 5 **should** be omitted.

The values of *highest_supported_security_strength* and *min_length* are provided in Table 2 of Section 10.1. The contents of the internal state are provided in Section 10.1.1.1.

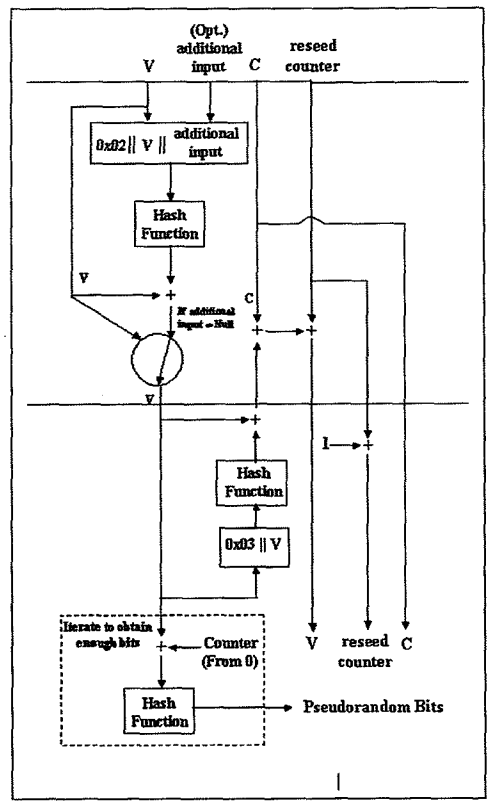

Figure 8: Hash_DRBG

The instantiate algorithm:

Let **Hash_df** be the hash derivation function specified in Section ~~9.5~~10.4.1 using the selected hash function. The output block length (*outlen*), seed length (*seedlen*) and appropriate *security_strengths* for the implemented hash function are provided in Table 2 of Section 10.1.

The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG (see step 9 of the instantiate process in Section 9.1).

Input:

1. *entropy_input*: The string of bits obtained from the entropy input source.
2. *nonce*: A string of bits as specified in Section 8.6.7.

3. *personalization_string*: The personalization string received from the consuming application. If a *personalization_string* will never be used, then step 1 may be modified to remove the *personalization_string*.

Output:

1. *initial_working_state*: The inital values for *V*, *C*, ~~*previous_output_block*~~ and *reseed_counter* (see Section 10.1.1.1).

Hash_DRBG Instantiate Process:

1. *seed_material* = *entropy_input* || *nonce* || *personalization_string*.
2. *seed* = **Hash_df** (*seed_material*, *seedlen*).
3. *V* = *seed*.
4. *C* = **Hash_df** ((0x00 || *V*), *seedlen*). Comment: Preceed *V* with a byte of zeros.
5. *reseed_counter* = 1.

~~Comment: Generate the initial block for comparing with the first DRBG output block (for continuous testing).~~

6. ~~*previous_output_block* = **Hash** (*V*).~~
7. ~~*H* = **Hash** (0x03 || *V*).~~
8. ~~*V* = (*V* + *H* + *C* + *reseed_counter*) mod $2^{seedlen}$.~~
9. ~~*reseed_counter* = 2.~~
10. ~~**Return** *V*, *C*, *previous_output_block* and~~ *reseed_counter* as the *initial_working_state*.

10.1.1.3 Reseeding a Hash_DRBG Instantiation

Notes for the reseed function specified in Section 9.2:

The reseeding of a **Hash_DRBG** instantiation requires a call to the reseed function. Process step 5 of that function calls the reseed algorithm specified in this section. The values for *min_length* are provided in Table 2 of Section 10.1.

The reseed algorithm:

Let **Hash_df** be the hash derivation function specified in Section ~~9.5~~10.4.1 using the selected hash function. The value for *seedlen* is provided in Table 2 of Section 10.1.

The following process or its equivalent **shall** be used as the reseed algorithm for this DRBG (see step 5 of the reseed process in Section 9.2):

Input:

1. *working_state*: The current values for *V*, *C*, ~~*previous_output_block*~~ and *reseed_counter* (see Section 10.1.1.1).
2. *entropy_input*: The string of bits obtained from the entropy input source.
3. *additional_input*: The additional input string received from the consuming application. If *additional_input* will never be provided, then step 1 may be modified to remove the *additional_input*.

Output:

1. *new_working_state*: The new values for *V*, *C*, ~~*previous_output_block*~~ and *reseed counter*.

Hash_DRBG Reseed Process:

1. *seed_material* = 0x01 || *V* || *entropy_input* || *additional_input*.
2. *seed* = **Hash_df** (*seed_material*, *seedlen*).
3. *V* = *seed*.
4. *C* = **Hash_df** ((0x00 || *V*), *seedlen*). Comment: Preceed with a byte of all zeros.
5. *reseed_counter* = 1.
6. Return *V*, *C*, ~~*previous_output_block*~~ and *reseed_counter* for the *new_working_state*.

10.1.1.4 Generating Pseudorandom Bits Using Hash_DRBG

Notes for the generate function specified in Section 9.3:

The generation of pseudorandom bits using a **Hash_DRBG** instantiation requires a call to the generate function. Process step 8 of that function calls the generate algorithm specified in this section. The values for *max_number_of_bits_per_request* and *outlen* are provided in Table 2 of Section 10.1.

The generate algorithm:

Let **Hash** be the selected hash function. The seed length (*seedlen*) and the maximum interval between reseeding (*reseed_interval*) are provided inTable 2 of Section 10.1. Note that for this DRBG, the reseed counter is used to update the value of *V* as well as to count the number of generation requests.

The following process or its equivalent **shall** be used as the generate algorithm for this DRBG (see step 8 of the generate process in Section 9.3):

Input:

1. *working_state*: The current values for *V*, *C*, ~~*previous_output_block*~~ and *reseed_counter* (see Section 10.1.1.1).

2. *requested_number_of_bits*: The number of pseudorandom bits to be returned to the generate function.
3. *additional_input*: The additional input string received from the consuming application. If *additional_input* will never be provided, then step 3 of the **Hash_DRBG** generate process may be omitted.

Output:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS, ~~ERROR,~~** or indicate that a reseed is required before the requested pseudorandom bits can be generated.
2. *returned_bits*: The pseudorandom bits to be returned to the generate function.
3. *new_working_state*: The new values for *V*, *C*, ~~*previous_output_block*~~ and *reseed_counter*.

Hash_DRBG Generate Process:

1. If *reseed_counter* > *reseed_interval*, then return an indication that a reseed is required.
2. If (*additional_input* ≠ *Null*), then do

   ~~3~~2.1 *w* = **Hash** (0x02 || *V* || *additional_input*).

   ~~3~~2.2 *V* = (*V* + *w*) mod $2^{seedlen}$.

4~~3~~. (~~*status, returned_bits, previous_output_block*~~) = **Hashgen** (*requested_number_of_bits*, *V*~~, previous_output_block~~).

5~~4~~. ~~If an **ERROR** is returned in step 4, then return **ERROR**.~~

~~6. *H* = **Hash** (0x03 || *V*).~~

7~~5~~. *V* = (*V* + *H* + *C* + *reseed_counter*) mod $2^{seedlen}$.

8~~6~~. *reseed_counter* = *reseed_counter* + 1.

9~~7~~. Return **SUCCESS**, ~~**SUCCESS,**~~ *returned_bits*, and the new values of *V*, *C*, ~~*previous_output_block*~~ and *reseed_counter* for the ~~*new*~~ *new_working_state*.

Hashgen (...):

Input:

1. *requested_no_of_bits*: The number of bits to be returned.
2. *V*: The current value of *V*.

Output:

~~3. *previous_output_block*: The last output block from the previous generate request.~~

~~**Output:**~~

1. ~~*status*: The status returned from this routine. The *status* will indicate **SUCCESS** or an **ERROR**.~~
2. ~~*returned_bits*~~: The generated bits to be returned to the generate function.

Hashgen Process:

3. ~~*previous_output_block*: The last output block generated using this routine.~~

~~**Process:**~~

1. $m = \left\lceil \frac{requested\_no\_of\_bits}{outlen} \right\rceil$.
2. $data = V$.
3. W = the *Null* string.
4. For $i = 1$ to m

   4.1 w_i = **Hash** ($data$).

   ~~4.2 If (w_i = *previous_output_block*), then return an **ERROR**.~~

   ~~4.3 *previous_output_block* = w_i.~~

   4.42 $W = W \| w_i$.

   4.53 $data = (data + 1) \bmod 2^{seedlen}$.
5. *returned_bits* = Leftmost (*requested_no_of_bits*) bits of W.
6. Return ~~**SUCCESS**~~, *returned_bits*~~, *previous_output_block*~~.

10.1.2 HMAC_DRBG (...)

HMAC_DRBG uses multiple occurrences of an Approved keyed hash function, which is based on an Approved hash function. This DRBG uses the **Update** function specified in Section 10.1.2.2 and the **HMAC** function within the **Update** function as the derivation function during instantiation and reseeding. The same hash function **shall** be used throughout. The hash function used **shall** meet or exceed the security requirements of the consuming application.

Figure 9 depicts the **HMAC_DRBG** in three stages. **HMAC_DRBG** is specified using an internal function **(Update)**. This function is called during the **HMAC_DRBG** instantiate, generate and reseed algorithms to adjust the internal state when new entropy or additional input is provided, as well as to update the internal state after pseudorandom bits are generated. The operations in the top portion of the figure are only performed if the additional input is not null. Figure 10 depicts the **Update** function.

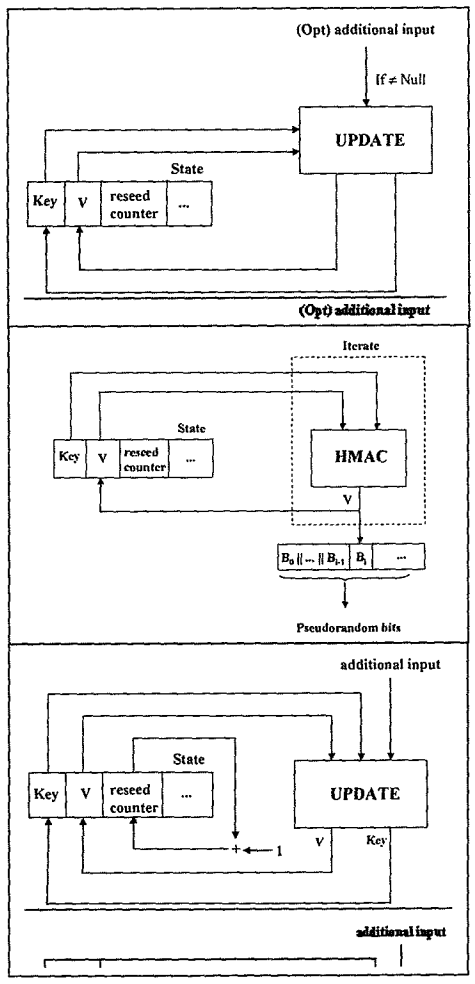

Figure 9: HMAC_DRBG Generate Function

10.1.2.1 HMAC_DRBG Internal State

The internal state for **HMAC_DRBG** consists of:

1. The *working_state*:
   a. The value *V* of *outlen* bits, which is updated each time another *outlen* bits of output are produced (where *outlen* is specified in Table 2 of Section 10.1).
   b. The *Key* of *outlen* bits, which is updated at least once each time that the DRBG generates pseudorandom bits.
   c. A counter (*reseed_counter*) that indicates the number of requests for pseudorandom bits since instantiation

or reseeding.

2. Administrative information:
   a. The *security_strength* of the DRBG instantiation.
   b. A *prediction_resistance_flag* that indicates whether or not a prediction resistance capability is required for the DRBG.

The values of *V* and *Key* are the critical values of the internal state upon which the security of this DRBG depends (i.e., *V* and *Key* are the "secret values" of the internal state).

10.1.2.2 The Update Function (Update)

The **Update** function updates the internal state of **HMAC_DRBG** using the *provided_data*. Note that for this DRBG, the **Update** function also serves as a derivation function for the instantiate and reseed functions.

Let **HMAC** be the keyed hash function specified in FIPS 198 using the hash function selected for the DRBG from Table 2 in Section 10.1.

The following or an equivalent process **shall** be used as the **Update** function.

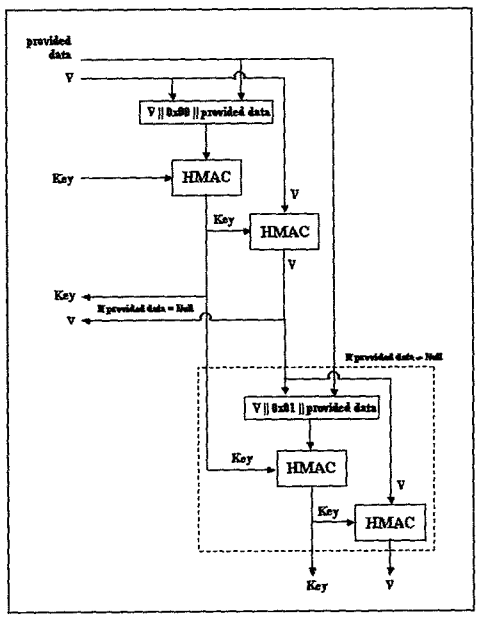

Figure 10: HMAC_DRBG Update Function

Input:

1. *provided_data*: The data to be used.
2. *K*: The current value of *Key*.
3. *V*: The current value of *V*.

Output:

1. *K*: The new value for *Key*.
2. *V*: The new value for *V*.

HMAC_DRBG Update Process:

1. *K* = **HMAC** (*K*, *V* || 0x00 || *provided_data*).
2. *V* = **HMAC** (*K*, *V*).
3. If (*provided_data* = *Null*), then return *K* and *V*.
4. *K* = **HMAC** (*K*, *V* || 0x01 || *provided_data*).
5. *V* = **HMAC** (*K*, *V*).

6. Return *K* and *V*.

10.1.2.3 Instantiation of HMAC_DRBG

Notes for the instantiate function specified in Section 9.1:

The instantiation of **HMAC_DRBG** requires a call to the instantiate function. Process step 9 of that function calls the instantiate algorithm specified in this section. For this DRBG, step 5 **should** be omitted. The values of *highest_supported_security_strength* and *min _length* are provided in Table 2 of Section 10.1. The contents of the internal state are provided in Section 10.1.2.1.

The instantiate algorithm:

Let **Update** be the function specified in Section 10.1.2.2. The ouput block length (*outlen*) is provided in Table 2 of Section 10.1.

The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG (see step 98 of the instantiate process in Section 9.1):

Input:

1. *entropy_input*: The string of bits obtained from the entropy input source.
2. *nonce*: A string of bits as specified in Section 8.6.7.
3. *personalization_string*: The personalization string received from the consuming application. If a *personalization_string* will never be used, then step 1 may be modified to remove the *personalization_string*.

Output:

1. *initial_working_state*: The inital values for *V*, *Key* and *reseed_counter* (see Section 10.1.2.1).

HMAC_DRBG Instantiate Process:

1. *seed_material* = *entropy_input* || *nonce* || *personalization_string*.
2. *Key* = 0x00 00...00. Comment: *outlen* bits.
3. *V* = 0x01 01...01. Comment: *outlen* bits.

   Comment: Update *Key* and *V*.
4. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).

   ~~Comment: Generate the initial block for comparing with the first DRBG output block (for continuous testing).~~

~~5. *V* = **HMAC** (*Key*, *V*).~~

~~6. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*)~~

~~7~~5. *reseed_counter* = 1.

~~8~~6. Return *V*, *Key* and *reseed_counter* as the ~~*initial*~~ *initial_working_state*.

10.1.2.4 Reseeding an HMAC_DRBG Instantiation

Notes for the reseed function specified in Section 9.2:

The reseeding of an **HMAC_DRBG** instantiation requires a call to the reseed function. Process step 5 of that function calls the reseed algorithm specified in this section. The values for *min_length* are provided in Table 2 of Section 10.1.

The reseed algorithm:

Let **Update** be the function specified in Section 10.1.2.2. The following process or its equivalent **shall** be used as the reseed algorithmn for this DRBG (see step 5 of the reseed process in Section 9.2):

Input:

1. *working_state*: The current values for *V*, *Key* and *reseed_counter* (see Section 10.1.2.1).
2. *entropy_input*: The string of bits obtained from the entropy input source.
3. *additional_input*: The additional input string received from the consuming application. If *additional_input* will never be used, then process step 1 may be modified to remove the *additional_input*.

Output:

1. *new_working_state*: The new values for *V*, *Key* and *reseed_counter*.

Process:

1. *seed_material* = *entropy_input* || *additional_input*.
2. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).
3. *reseed_counter* = 1.
4. **Return** *V*, *Key* and *reseed_counter* as the ~~*new*~~ *new_working_state*.

10.1.2.5 Generating Pseudorandom Bits Using HMAC_DRBG

Notes for the generate function specified in Section 9.3:

The generation of pseudorandom bits using an **HMAC_DRBG** instantiation requires a call to the generate function. Process step 8 of that function calls the generate algorithm specified in this section. The values for *max_number_of_bits_per_request* and *outlen* are provided in Table 2 of Section 10.1.

The generate algorithm :

Let **HMAC** be the keyed hash function specified in FIPS 198 using the hash function

selected for the DRBG. The value for *reseed_interval* is defined in Table 2 of Section 10.1.

The following process or its equivalent **shall** be used as the generate algorithm for this DRBG (see step 8 of the generate process in Section 9.3):

Input:

1. *working_state*: The current values for *V*, *Key* and *reseed_counter* (see Section 10.1.2.1).
2. *requested_number_of_bits*: The number of pseudorandom bits to be returned to the generate function.
3. *additional_input*: The additional input string received from the consuming application. If an implementation will never use *additional_input*, then step 3 of the HMAC generate process may be omitted. If an implementation does not include the *additional_input* parameter as one of the calling parameters, or if the implementation allows *additional_input*, but a given request does not provide any *additional_input*, then a *Null* string **shall** be used as the *additional_input* in step 6.

Output:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS,** ~~an **ERROR**~~ or indicate that a reseed is required before the requested pseudorandom bits can be generated.
2. *returned_bits*: The pseudorandom bits to be returned to the generate function.
3. *new_working_state*: The new values for *V*, *Key* and *reseed_counter*.

HMAC_DRBG Generate Process:

1. If *reseed_counter* > *reseed_interval*, then return an indication that a reseed is required.
2. ~~*V_old = V.*~~
3. ~~If *additional_input* ≠ *Null*, then (*Key*, *V*) = **Update** (*additional_input*, *Key*, *V*).~~
4. *temp* = *Null*.
5. While (**len** (*temp*) < *requested_number_of_bits*) do:

   ~~5~~.1 *V* = **HMAC** (*Key*, *V*).

   ~~Comment: Continuous test – check that successive values of *V* are not identical.~~

   ~~5.2 If (*V = V_old*), then return an **ERROR**.~~

   ~~5.3 *V_old = V.*~~

~~5.4~~4.2 *temp* = *temp* || *V*.

~~6~~5. *returned_bits* = Leftmost *requested_number_of_bits* of *temp*.

~~7~~6. (*Key*, *V*) = **Update** (*additional_input*, *Key*, *V*).

~~8~~7. *reseed_counter* = *reseed_counter* + 1.

~~9~~8. Return **SUCCESS**, *returned_bits*, and the new values of *Key*, *V* and *reseed_counter* as the *new_working_state*).

10.2 DRBGs Based on Block Ciphers

A block cipher DRBG is based on a block cipher algorithm. The block cipher DRBG specified in this Recommendation has been designed to use any Approved block cipher algorithm and may be used by consuming applications requiring various levels of security, providing that the appropriate block cipher algorithm and key length are used, and sufficient entropy is obtained for the seed.

10.2.1 CTR_DRBG

CTR_DRBG uses an Approved block cipher algorithm in the counter mode as specified in SP 800-38A. The same block cipher algorithm and key length **shall** be used for all block cipher operations. The block cipher algorithm and key length **shall** meet or exceed the security requirements of the consuming application.

CTR_DRBG is specified using an internal function (**Update**). Figure 11 depicts the **Update** function. This function is called by the instantiate, generate and reseed algorithms to adjust the internal state when new entropy or additional input is provided, as well as to update the internal state after pseudorandom bits are generated. Figure 12 depicts the **CTR_DRBG** in three stages. The operations in the top portion of the figure are only performed if the additional input is not null.

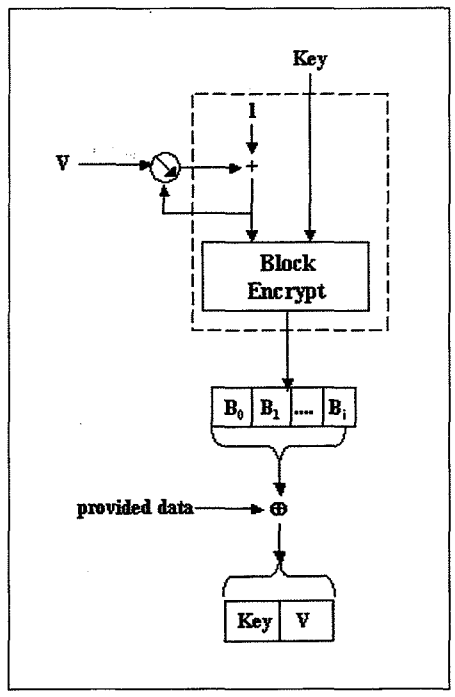

Figure 11: CTR_DRBG Update Function

Table 3 specifies the values that **shall** be used for the function envelopes and DRBG algorithms.

Table 3: Definitions for the CTR_DRBG

	3 Key TDEA	AES-128	AES-192	AES-256
Supported security strengths	See SP 800-57			
highest_supported_security_strength	See SP 800-57			
Output block length (*outlen*)	64	128	128	128
Key length (*keylen*)	168	128	192	256

	3 Key TDEA	AES-128	AES-192	AES-256
Required minimum entropy for instantiate and reseed	*security_strength*			
Seed length (*seedlen = outlen + keylen*)	232	256	320	384
If a derivation function is used:				
Minimum entropy input length (*min _length*)	*security_strength*			
Maximum entropy input length (*max _length*)	$\leq 2^{35}$ bits			
Maximum personalization string length (*max_personalization_string_length*)	$\leq 2^{35}$ bits			
Maximum additional_input length (*max_additional_input_length*)	$\leq 2^{35}$ bits			
If a derivation function is not used (full entropy is available):				
Minimum entropy input length (*min _length*) (= *outlen + keylen*)	*seedlen*			
Maximum entropy input length (*max _length*) (*outlen + keylen*)	*seedlen*			
Maximum personalization string length (*max_personalization_string_length*)	*seedlen*			
Maximum additional_input length (*max_additional_input_length*)	*seedlen*			
max_number_of_bits_per_request	$\leq 2^{13}$	$\leq 2^{19}$		
Number of requests between reseeds (*reseed_interval*)	$\leq 2^{32}$	$\leq 2^{48}$		

The CTR_ DRBG may be implemented to use the block cipher derivation function specified in Section 9.510.4.2 during instantiation and reseeding. However, the DRBG is specified to allow an implementation tradeoff with respect to the use of this derivation function. If a source for full entropy input is always available to provide entropy input when requested, the use of the derivation function is optional; otherwise, the derivation functon **shall** be used. Table 3 provides lengths required for the *entropy_input*,

personalization_string and *additional_input* for each case.

When full entropy is available, and a derivation function is not used by an implementation, the seed construction (see Section 8.6.1) **shall not** use a nonce[4].

When using TDEA as the selected block cipher algorithm, the keys **shall** be handled as 64-bit blocks containing 56 bits of key and 8 bits of parity as specified for the TDEA engine specified in SP 800-67.

10.2.1.1 CTR_DRBG Internal State

The internal state for **CTR_DRBG** consists of:

1. The *working_state*:
   a. The value *V* of *outlen* bits, which is updated each time another *outlen* bits of output are produced.
   b. The *Key* of *keylen* bits, which is updated whenever a predetermined number of output blocks are generated.
   c. ~~The *previous_output_block*; this is required to perform a continuous test on the output from the generate function.~~
   ~~d.~~ A counter (*reseed_counter*) that indicates the number of

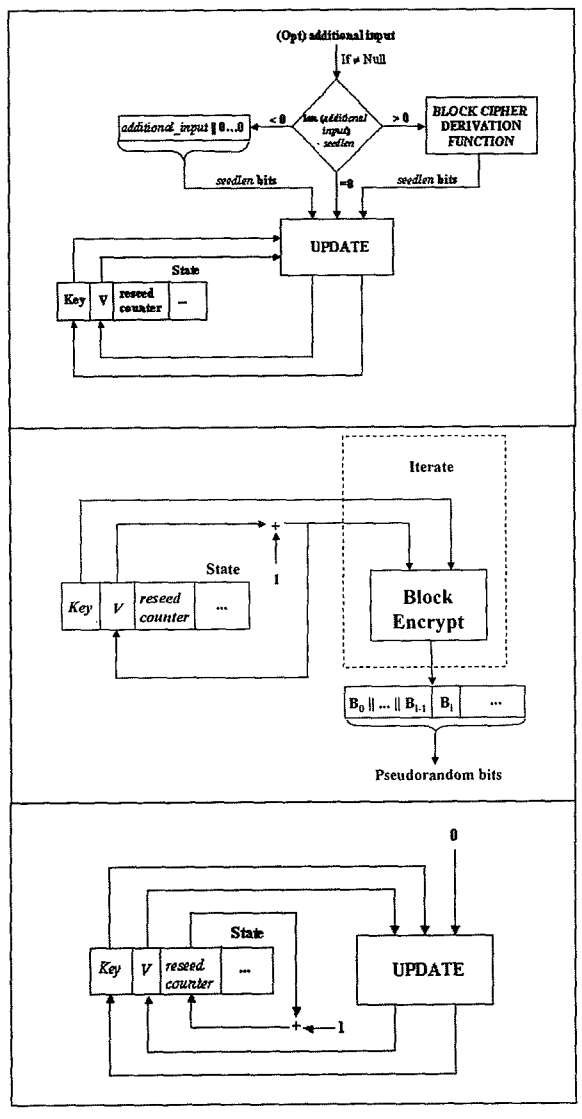

Figure 12: CTR-DRBG

[4] The specifications in this Standard do not accommodate the special treatment required for a nonce in this case.

requests for pseudorandom bits since instantiation or reseeding.

2. Administrative information:
   a. The *security_strength* of the DRBG instantiation.
   b. A *prediction_resistance_flag* that indicates whether or not a prediction resistance capability is required for the DRBG.

The values of *V* and *Key* are the critical values of the internal state upon which the security of this DRBG depends (i.e., *V* and *Key* are the "secret values" of the internal state).

10.2.1.2 The Update Function (Update)

The **Update** function updates the internal state of the **CTR_DRBG** using the *provided_data*. The values for *outlen*, *keylen* and *seedlen* are provided in Table 3 of Section 10.2.1. The block cipher operation in step 2.2 of the **CTR_DRBG** update process uses the selected block cipher algorithm (also see Section 10.4).

The following or an equivalent process **shall** be used as the **Update** function.

Input:

1. *provided_data*: The data to be used. This must be exactly *seedlen* bits in length; this length is guaranteed by the construction of the *provided_data* in the instantiate, reseed and generate functions.
2. *Key*: The current value of *Key*.
3. *V*: The current value of *V*.

Output:

1. *K*: The new value for *Key*.
2. *V*: The new value for *V*.

CTR_DRBG Update Process:

1. *temp* = *Null*.
2. While (**len** (*temp*) < *seedlen*) do
   2.1 $V = (V + 1) \bmod 2^{outlen}$.
   2.2 *output_block* = **Block_Encrypt** (*Key*, *V*).
   2.3 *temp* = *temp* || *ouput_block*.
3. *temp* = Leftmost *seedlen* bits of *temp*.
4 *temp* = *temp* ⊕ *provided_data*.
5. *Key* = Leftmost *keylen* bits of *temp*.

6. *V* = Rightmost *outlen* bits of *temp.*
7. Return the new values of *Key* and V.

10.2.1.3 Instantiation of CTR_DRBG

Notes for the instantiate function specified in Section 9.1:

The instantiation of **CTR_DRBG** requires a call to the instantiate function. Process step 9 of that function calls the instantiate algorithm specified in this section. For this DRBG, step 5 **should** be omitted. The values of *highest_supported_security_strength* and *min _length* are provided in Table 3 of Section 10.2.1. The contents of the internal state are provided in Section 10.2.1.1.

The instantiate algorithm:

Let **Update** be the function specified in Section 10.2.1.2~~, and let **Block_Cipher_df** be the derivation function specified in Section 9.5.2 using the chosen block cipher algorithm and key size~~. The output block length (*outlen*), key length (*keylen*), seed length (*seedlen*) and *security_strengths* for the block cipher algorithms are provided in Table 3 of Section 10.2.1.

For this DRBG, there are two cases for the processing. The input to the instantiate algorithm is the same for each case; likewise for the output from the instantiate algorithm. However, the process steps are slightly different (see Sections 10.2.1.3.1 and 10.2.1.3.2). ~~The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG:~~

Input:

1. *entropy_input*: The string of bits obtained from the entropy input source.
2. *nonce*: A string of bits as specified in Section 8.6.7; this string **shall not** be present when a derivation function is not used.
3. *personalization_string*: The personalization string received from the consuming application.

Output:

1. *initial_working_state*: The inital values for *V*, *Key*, ~~*previous_output_block*~~ and *reseed_counter* (see Section 10.2.1.1).

10.2.1.3.1 The Process Steps for Instantiation When Full Entropy is Available for the Entropy Input, and a Derivation Function is Not Used

The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG:

CTR_DRBG Instantiate Process:

1. ~~*If the block cipher derivation function is available, then*~~

~~*1.1 seed_material = entropy_input || nonce || personalization_string.*~~

~~*1.2 seed_material =* **~~Block_Cipher_df~~** *(seed_material, seedlen).*~~

~~*Else -- Comment: The block cipher derivation function is not used and full entropy is known to be available.*~~

~~*1.3*~~ *temp* = **len** (*personalization_string*).

Comment: Ensure that the length of the *personalization_string* is exactly *seedlen* bits. The maximum length was checked in Section 9.1, processing step 3, using Table 3 to define the maximum length.

2. ~~1.4~~ If (*temp* < *seedlen*), then *personalization_string* = *personalization_string* || $0^{seedlen - temp}$.

3. ~~1.5~~ *seed_material* = *entropy_input* ⊕ *personalization_string*.

~~2~~4. *Key* = 0^{keylen}. Comment: *keylen* bits of zeros.

~~3~~5. *V* = 0^{outlen}. Comment: *outlen* bits of zeros.

~~4~~6. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).

~~5~~7. *reseed_counter* = 1.

~~Comment: Generate the initial block for comparing with the first DRBG output block (for continuous testing).~~

~~6. *previous_output_block* = **Block_Encrypt** (*Key*, *V*).~~

~~7. *zeros* = $0^{seedlen}$. Comment: Produce a string of *seedlen* zeros.~~

~~8. (*Key*, *V*) = **Update** (*zeros*, *Key*, *V*).~~

~~9~~8. **Return** *V*, *Key*, ~~*previous_output_block*~~ and *reseed_counter* as the *initial_working_state*.

Implementation note:

If a *personalization_string* will never be provided from the instantiate function, then steps 1-3 are replaced by:

seed_material = entropy_input.

That is, steps 1-3 collapse into the above step.

10.2.1.3.2 The Process Steps for Instantiation When a Derivation Function is Used

Let **Block_Cipher_df** be the derivation function specified in Section 10.4.2 using the

chosen block cipher algorithm and key size

The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG:

CTR_DRBG Instantiate Process:

1. *seed_material* = *entropy_input* || *nonce* || *personalization_string*.

   Comment: Ensure that the length of the *seed_material* is exactly *seedlen* bits.

2. *seed_material* = **Block_Cipher_df** (*seed_material*, *seedlen*).
3. *Key* = 0^{keylen}. Comment: *keylen* bits of zeros.
4. *V* = 0^{outlen}. Comment: *outlen* bits of zeros.
5. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).
6. *reseed_counter* = 1.
7. **Return** *V*, *Key*, and *reseed_counter* as the *initial_working_state*.

Implementation note:~~s~~:

~~1.~~ If a *personalization_string* will never be provided from the instantiate function~~ and a derivation function will be used~~, then steps 1-~~2.1 becomes~~are replaced by:

seed_material = **Block_Cipher_df** (*entropy_input*, *seedlen*).

~~2. If a *personalization_string* will never be provided from the instantiate function, a full entropy source will be available and a derivation function will not be used, then step 1 becomes~~

~~*seed_material = entropy_input.*~~

~~That is, steps 1.3 – 1.6 collapse into the above step.~~

10.2.1.4 Reseeding a CTR_DRBG Instantiation

Notes for the reseed function specified in Section 9.2:

The reseeding of a **CTR_DRBG** instantiation requires a call to the reseed function. Process step 5 of that function calls the reseed algorithm specified in this section. The values for *min _length* are provided in Table 3 of Section 10.2.1.

The reseed algorithm:

Let **Update** be the function specified in Section 10.2.1.2~~, and let **Block_Cipher_df** be the derivation function specified in Section 9.5.2 using the chosen block cipher algorithm and key size~~. The seed length (*seedlen*) is provided in Table 3 of Section 10.2.1.

For this DRBG, there are two cases for the processing. The input to the reseed algorithm is the same for each case; likewise for the output from the reseed algorithm. However, the process steps are slightly different (see Sections 10.2.1.4.1 and 10.2.1.4.2).

~~If a block cipher derivation function is to be used, then the **Block_Cipher_df** specified in Section 9.5.2 **shall** be implemented using the chosen block cipher algorithm and key size; in this case, step 1 below **shall** consist of steps 1.1 and 1.2 (i.e., steps 1.3 to 1.5 **shall not** be used).~~

~~If full entropy is available whenever entropy input is required, and a block cipher derivation function is not to be used, then step 1 below **shall** consist of steps 1.3 to 1.5 (i.e., steps 1.1 and 1.2 **shall not** be used).~~

~~The following process or its equivalent **shall** be used as the reseed algorithm for this DRBG (see step 5 of Section 9.2):~~

Input:

1. *working_state*: The current values for *V*, *Key* and *reseed_counter* (see Section 10.2.1.1).
2. *entropy_input*: The string of bits obtained from the entropy input source.
3. *additional_input*: The additional input string received from the consuming application.

Output:

1. *new_working_state*: The new values for *V*, *Key*, ~~*previous_output_block*~~ and *reseed_counter*.

10.2.1.4.1 The Process Steps for Reseeding When Full Entropy is Available for the Entropy Input, and a Derivation Function is Not Used

The following process or its equivalent **shall** be used as the reseed algorithm for this DRBG (see step 5 of the reseed process in Section 9.2):

CTR_DRBG Reseed

Process:

~~*If the block cipher derivation function is available, then*~~

~~*seed_material = entropy_input || additional_input.*~~

~~*seed_material = **Block_Cipher_df** (seed_material, seedlen).*~~

~~*Else* *Comment: The block cipher derivation function is not used because full entropy is known to be available.*~~

~~1.3~~ *temp* = **len** (*additional_input*).

Comment: Ensure that the length of the *additional_input* is exactly *seedlen* bits. The maximum length was checked in Section 9.2, processing step 2, using Table 3 to define the maximum length.

2. ~~1.4~~ If (*temp* < *seedlen*), then *additional_input* = *additional_input* || $0^{seedlen - temp}$.

3. ~~1.5~~ *seed_material* = *entropy_input* ⊕ *additional_input*.

~~2~~4. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).

~~3~~5. *reseed_counter* = 1.

~~4~~6. **Return** *V*, *Key*, ~~*previous_output_block*~~ and *reseed_counter* as the *new_working_state*.

Implementation note:

If *additional_input* will never be provided from the reseed function, then steps 1-3 are replaced by:

seed_material = *entropy_input*.

That is, steps 1-3 collapse into the above step.

10.2.1.4.2 The Process Steps for Reseeding When a Derivation Function is Used

Let **Block_Cipher_df** be the derivation function specified in Section 10.4.2 using the chosen block cipher algorithm and key size.

The following process or its equivalent **shall** be used as the reseed algorithm for this DRBG (see step 5 of Section 9.2):

CTR_DRBG Reseed Process:

1. *seed_material* = *entropy_input* || *additional_input*.

Comment: Ensure that the length of the *seed_material* is exactly *seedlen* bits.

2. *seed_material* = **Block_Cipher_df** (*seed_material*, *seedlen*).

3. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).

4. *reseed_counter* = 1.

5. **Return** *V*, *Key*, and *reseed_counter* as the *new_working_state*.

Implementation notes:

~~1.~~ If *additional_input* will never be provided from the reseed function ~~and a~~

derivation function will be used, then steps 1-2 1.1 becomes:

seed_material = **Block_Cipher_df** (*entropy_input*, *seedlen*).

2. If *additional_input* will never be provided from the reseed function, a full entropy source will be available and a derivation function will not be used, then step 1 becomes

seed_material = entropy_input.

That is, steps 1.3 – 1.6 collapse into the above step.

10.2.1.5 Generating Pseudorandom Bits Using CTR_DRBG

Notes for the generate function specified in Section 9.3:

The generation of pseudorandom bits using a **CTR_DRBG** instantiation requires a call to the generate function. Process step 8 of that function calls the generate algorithm specified in this section. The values for *max_number_of_bits_per_request* and *max_additional_input_length*, and *outlen* are provided in Table 3 of Section 10.2.1.

For this DRBG, there are two cases for the processing. The input to the generate algorithm is the same for each case; likewise for the output from the generate algorithm. However, the process steps are slightly different (see Sections 10.2.1.5.1 and 10.2.1.5.2).

If the derivation function is not used, then the maximum allowed length of *additional_input = seedlen.*

Let **Update** be the function specified in Section 10.2.1.2, and let **Block_Encrypt** be the function specified in Section 10.4.2. The seed length (*seedlen*) and the value of *reseed_interval* are provided in Table 3 of Section 10.2.1.

Step 5.2 below uses the selected block cipher algorithm. If a derivation function is not used for a DRBG implementation, then step 3.2 **shall** be omitted.

If a block cipher derivation function is to be used, then the **Block_Cipher_df** specified in Section 9.5.2 **shall** be implemented using the chosen block cipher algorithm and key size; in this case, step 3.2 below **shall** be included.

If full entropy is available whenever entropy input is required, and a block cipher derivation function is not to be used, then step 3.2 below **shall not** be used.

The following process or its equivalent **shall** be used as generate algorithm for this DRBG (see step 8 of Section 9.3):

Input:

1. *working_state*: The current values for *V*, *Key*, *previous_output_block* and *reseed_counter* (see Section 10.2.1.1).
2. *requested_number_of_bits*: The number of pseudorandom bits to be returned

to the generate function.

3. *additional_input*: The additional input string received from the consuming application. If *additional_input* will never be allowed, then step 3 becomes:

$$additional\_input = 0^{seedlen}.$$

Output:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS,** ~~an **ERROR**~~ or indicate that a reseed is required before the requested pseudorandom bits can be generated.
2. *returned_bits*: The pseudorandom bits returned to the generate function.
3. *working_state*: The new values for *V*, *Key*, ~~*previous_output_block*~~ and *reseed_counter*.

10.2.1.5.1 The Process Steps for Generating Pseudorandom BitsWhen a Derivation Function is Not Used for the DRBG Implementation

The following process or its equivalent **shall** be used as the generate algorithm for this DRBG (see step 8 of the generate process in Section 9.3.3):

CTR_DRBG Generate

Process:

1. If *reseed_counter* > *reseed_interval*, then return an indication that a reseed is required.
2. ~~*V_old = V.*~~
3. ~~If~~ (*additional_input* ≠ *Null*), then

   Comment: Ensure that the length of the *additional_input* is exactly *seedlen* bits. The maximum length was checked in Section 9.3.3, processing step 4, using Table 3 to define the maximum length. If the length of the *additional input* is ~~>~~ *seedlen*, ~~derive *seedlen*~~pad with zero ~~b~~bits.

   ~~3~~2.1 *temp* = **len** (*additional_input*).

   ~~Comment: If a block cipher derivation function is used:~~

   ~~3.2 If (*temp* > *seedlen*), then *additional_input* = **Block_Cipher_df** (*additional_input*, *seedlen*).~~

   ~~Comment: If the length of the *additional_input* is < *seedlen*, pad with zeros to *seedlen* bits.~~

   ~~3~~2.~~3~~2 If (*temp* < *seedlen*), then

additional_input = *additional_input* || $0^{seedlen - temp}$.

~~3~~2.4~~3~~ (*Key*, *V*) = **Update** (*additional_input*, *Key*, *V*).

Else *additional_input* = $0^{seedlen}$.

4~~3~~. *temp* = *Null*.

5~~4~~. While (**len** (*temp*) < *requested_number_of_bits*) do:

~~5~~4.1 *V* = (*V* + 1) mod 2^{outlen}.

~~5~~4.2 *output_block* = **Block_Encrypt** (*Key*, V).

~~Comment: Continuous test -- Check that the old and new output blocks are different.~~

~~5.3 If (*output_block* = *previous_output_block*), then return an **ERROR**.~~

~~5.4 *previous_output_block* = *output_block*.~~

~~5~~4.5~~3~~ *temp* = *temp* || *output_block*.

6~~5~~. *returned_bits* = Leftmost *requested_number_of_bits* of *temp*.

~~7~~Comment: Update for backtracking resistance.

~~8~~6. (*Key*, *V*) = **Update** (*additional_input*, *Key*, *V*).

~~9~~7. *reseed_counter* = *reseed_counter* + 1.

~~10~~8. Return **SUCCESS** and *returned_bits*; also return *Key*, *V*, ~~*previous_output_block*~~ and *reseed_counter* as the ~~*new*~~ *new_working_state*.

10.2.1.5.2 The Process Steps for Generating Pseudorandom BitsWhen a Derivation Function is Used for the DRBG Implementation

The **Block_Cipher_df** is specified in Section 10.4.2 and **shall** be implemented using the chosen block cipher algorithm and key size.

The following process or its equivalent **shall** be used as generate algorithm for this DRBG (see step 8 of the generate process in Section 9.3.3):

CTR_DRBG Generate Process:

1. If *reseed_counter* > *reseed_interval*, then return an indication that a reseed is required.
2. If (*additional_input* ≠ *Null*), then
   2.1 *additional_input* = **Block_Cipher_df** (*additional_input*, *seedlen*).
   2.2 (*Key*, *V*) = **Update** (*additional_input*, *Key*, *V*).

Else *additional_input* = $0^{seedlen}$.

3. *temp* = *Null*.
4. While (**len** (*temp*) < *requested_number_of_bits*) do:
   4.1 *V* = (*V* + 1) mod 2^{outlen}.
   4.2 *output_block* = **Block_Encrypt** (*Key*, V).
   4.3 *temp* = *temp* || *output_block*.
5. *returned_bits* = Leftmost *requested_number_of_bits* of *temp*.

   Comment: Update for backtracking resistance.

6. (*Key*, *V*) = **Update** (*additional_input*, *Key*, *V*).
7. *reseed_counter* = *reseed_counter* + 1.
8. Return **SUCCESS** and *returned_bits*; also return *Key*, *V*, and *reseed_counter* as the *new_working_state*.

10.3 Deterministic RBGs Based on Number Theoretic Problems

A DRBG can be designed to take advantage of number theoretic problems (e.g., the discrete logarithm problem). If done correctly, such a generator's properties of randomness and/or unpredictability will be assured by the difficulty of finding a solution to that problem. This section specifies a DRBG based on the elliptic curve discrete logarithm problem.

10.3.1 Dual Elliptic Curve Deterministic RBG (Dual_EC_DRBG)

Dual_EC_DRBG is based on the following hard problem, sometimes known as the "elliptic curve discrete logarithm problem" (ECDLP): given points P and Q on an elliptic curve of order n, find a such that $Q = aP$.

Dual_EC_DRBG uses a seed that is m bits in length (i.e., *seedlen* = m) to initiate the generation of *outlen*-bit pseudorandom strings by performing scalar multiplications on two points in an elliptic curve group, where the curve is defined over a field approximately 2^m in size. For all the NIST curves given in this Recommendation, $m \geq 256$~~163~~. Figure 13 depicts the **Dual_EC_DRBG**.

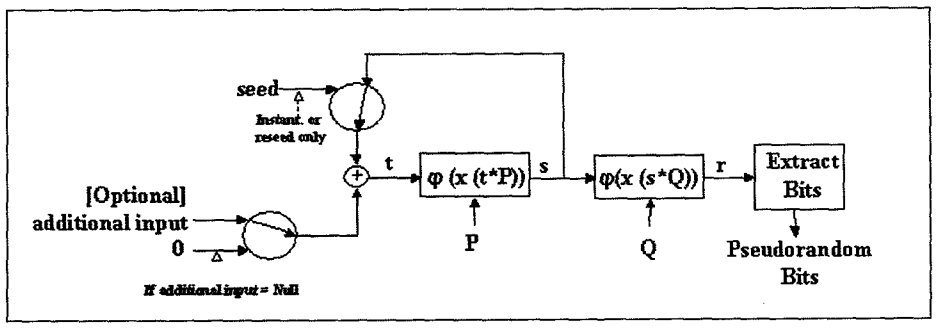

Figure 13: Dual_EC_DRBG

The instantiation of this DRBG requires the selection of an appropriate elliptic curve and curve points specified in Appendix A.1 for the desired security strength. The *seed* used to determine the initial value (s) of the DRBG **shall** have entropy that is at least *security_strength* bits. Further requirements for the *seed* are provided in Section 8.6. This DRBG uses the derivation function specified in Section 10.4.1 during instantiation and reseeding.

Backtracking resistance is inherent in the

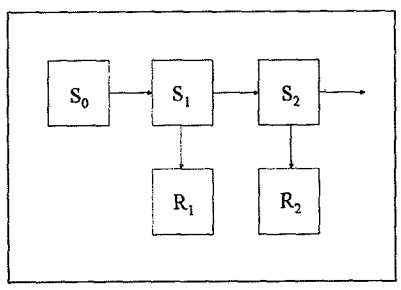

Figure 14: Dual_EC_DRBG (...) Backtracking Resistance

algorithm, even if the internal state is compromised. As shown in Figure 14, **Dual_EC_DRBG** generates a *seedlen*-bit number for each step $i = 1,2,3,\ldots$, as follows:

$S_i = \varphi(x(S_{i-1} * P))$

$R_i = \varphi(x(S_i * Q))$.

Each arrow in the figure represents an Elliptic Curve scalar multiplication operation, followed by the extraction of the x coordinate for the resulting point and for the random output R_i, followed by truncation to produce the output. Following a line in the direction of the arrow is the normal operation; inverting the direction implies the ability to solve the ECDLP for that specific curve. An adversary's ability to invert an arrow in the figure implies that the adversary has solved the ECDLP for that specific elliptic curve. Backtracking resistence is built into the design, as knowledge of S_1 does not allow an adversary to determine S_0 (and so forth) unless the adversary is able to solve the ECDLP for that specific curve. In addition, knowledge of R_1 does not allow an adversary to determine S_1 (and so forth) unless the adversary is able to solve the ECDLP for that specific curve.

Table 4 specifies the values that **shall** be used for the envelope and algorithm for each curve. Complete specifications for each curve are provided in Appendix A.1. Note that all curves ~~except the P-224 curve~~ can be instantiated at a security strength lower than its highest possible security strength. For example, the highest security strength that can be supported by curve P-384 is 192 bits; however, this curve can alternatively be instantiated to support only the 112 or 128-bit security strengths).

Table 4: Definitions for the Dual_EC_DRBG

	~~P-224~~ P-256	P-384	P-521
Supported security strengths	See SP 800-57		
Size of the base field (in bits)	256	384	521
highest_supported_ security_strength	See SP 800-57		
Output block length (*max_outlen* = largest multiple of 8 less than (size of the base field) - (13 + $\log_2$ (the cofactor))	~~208~~ 240	368	504
Required minimum entropy for instantiate and reseed	*security_strength*		
Minimum entropy input length (*min_length*)	*security_strength*		
Maximum entropy input length (*max _length*)	$\leq 2^{13}$ bits		

	~~P-224~~ P-256	P-384	P-521
Maximum personalization string length (*max_personalization_string_length*)	$\leq 2^{13}$ bits		
Maximum additional input length (*max_additional_input_length*)	$\leq 2^{13}$ bits		
Seed length (*seedlen*)	**2 × *security_strength***		
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		

10.3.1.1 Dual_EC_DRBG Internal State

The internal state for **Dual_EC_DRBG** consists of:

1. The *working_state*:
   a. A value (*s*) that determines the current position on the curve.
   b. The elliptic curve domain parameters (*seedlen*, *p*, *a*, *b*, *n*), where *seedlen* is the length of the seed ; *a* and *b* are two field elements that define the equation of the curve, and *n* is the order of the point *G*. If only one curve will be used by an implementation, these parameters need not be present in the *working_state*.
   c. Two points *P* and *Q* on the curve; the generating point *G* specified in FIPS 186-3 for the chosen curve will be used as *P*. If only one curve will be used by an implementation, these points need not be present in the *working_state*.
   d. ~~*r_old*, the previous output block.~~
   ~~e.~~ A counter (*block_counter*) that indicates the number of blocks of random produced by the **Dual_EC_DRBG** since the initial seeding or the previous reseeding.
2. Administrative information:
   a. The *security_strength* provided by the instance of the DRBG,
   b. A *prediction_resistance_flag* that indicates whether prediction resistance is required by the DRBG.

The value of *s* is the critical value of the internal state upon which the security of this DRBG depends (i.e., *s* is the "secret value" of the internal state).

10.3.1.2 Instantiation of Dual_EC_DRBG

Notes for the instantiate function specified in Section 9.1:

The instantiation of **Dual_EC_DRBG** requires a call to the instantiate function. Process step 9 of that function calls the instantiate algorithm in this section.

In process step 5 of the instantiate function, the following step **shall** be performed to select an appropriate curve if multiple curves are available.

5. Using the *security_strength* and Table 4 in Section 10.3.1, select the smallest available curve that has a security strength ≥ *security_strength*.

The values for *seedlen, p, a, b, n, P, Q* are determined by that curve.

It is recommended that the default values be used for *P* and *Q* as given in Appendix A.1. However, an implementation ~~may~~may use different pairs of points, provided that they are *verifiably random*, as evidenced by the use of the procedure specified in Appendix A.2.1 and the self-test procedure described in Appendix A.2.2.

The values for *highest_supported_security_strength* and *min_length* are determined by the selected curve (see Table 4 in Section 10.3.1).

The instantiate algorithm :

Let **Hash_df** be the hash derivation function specified in Section ~~9.5~~10.4.1 using an appropriate hash function from Table 4 in Section 10.3.1. Let *seedlen* be the appropriate value from Table 4.

The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG (see step 9 of the instantiate process in Section 9.1):

Input:

1. *entropy_input*: The string of bits obtained from the entropy input source.
2. *nonce*: A string of bits as specified in Section 8.6.7.
3. *personalization_string*: The personalization string received from the consuming application.

Output:

1. *s*: The initial secret value for the *initial_working_state*.
2. ~~*r_old*: The initial output block (which will not be used).~~
3. ~~*block_counter*~~: The initialized block counter for reseeding.

Dual_EC_DRBG Instantiate Process:

1. *seed_material* = *entropy_input* || *nonce* || *personalization_string*.

   Comment: Use a hash function to ensure that the entropy is distributed throughout the bits, and *s* is *m* (i.e., *seedlen*) bits in length.

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

3. ~~*r_old* = φ(*x*(*s* * *Q*)).~~ ~~Comment: *r* is a *seedlen* bit number.~~

~~4. *block_counter* = 0.~~

~~5~~4. Return *s*, ~~*r_old*~~ and *block_counter* for the *initial_working_state*.

10.3.1.3 Reseeding of a Dual_EC_DRBG Instantiation

Notes for the reseed function specified in Section 9.2:

The reseed of **Dual_EC_DRBG** requires a call to the reseed function. Process step 5 of that function calls the reseed algorithm in this section. The values for *min _length* are provided in Table 4 of Section 10.3. 1.

The reseed algorithm :

Let **Hash_df** be the hash derivation function specified in Section 10.~~49.6.12~~ using an appropriate hash function from Table 4 in Section 10.3. 1.

The following process or its equivalent **shall** be used to reseed the **Dual_EC_DRBG** process after it has been instantiated (see step 54 of the reseed process in Section 9.2):

Input:

1. *s*: The current value of the secret parameter in the *working_state*.
2. *entropy_input*: The string of bits obtained from the entropy input source.
3. *additional_input*: The additional input string received from the consuming application.

Output:

1. *s*: The new value of the secret parameter in the *new_working_state*.
2. *block_counter*: The re-initialized block counter for reseeding.

Dual_EC_DRBBG Reseed Process:

Comment: **pad8** returns a copy of *s* padded on the right with binary 0's, if necessary, to a multiple of 8.

1. *seed_material* = **pad8** (*s*) || *entropy_input* || *additional_input_string*.
2. *s* = **Hash_df** (*seed_material*, *seedlen*).

3. *block_counter* = 0.
4. Return *s* and *block_counter* for the *new-_working_state*.

Implementation notes:

If an implementation never allows *additional_input*, then step 1 may be modified as follows :

seed_material = **pad8** (*s*) || *entropy_input.*

10.3.1.4 Generating Pseudorandom Bits Using Dual_EC_DRBG

Notes for the generate function specified in Section 9.3:

The generation of pseudorandom bits using a **Dual_EC_DRBG** instantiation requires a call to the generate function. Process step 8 of that function calls the generate algorithm specified in this section. The values for *max_number_of_bits_per_request* and *max_outlen* are provided in Table 4 of Section 10.3.1. *outlen* is the number of pseudorandom bits taken from each *x*-coordinate as the **Dual_EC_DRBG** steps. For performance reasons, the value of *outlen* should be set to the maximum value as provided in Table 4. However, an implementation ~~may~~may set *outlen* to any multiple of 8 bits less than or equal to *max_outlen*. The bits that become the **Dual_EC_DRBG** output are always the rightmost bits, i.e., the least significant bits of the *x*-coordinates.

The generate algorithm:

Let **Hash_df** be the hash derivation function specified in Section ~~9.5~~10.4.1 using an appropriate hash function from Table 4 in Section 10.3.1. The value of *reseed_interval* is also provided in Table 4.

The following are used by the generate algorithm:

a. **pad8** (bitstring) returns a copy of the *bitstring* padded on the right with binary 0's, if necessary, to a multiple of 8.

b. **Truncate** (*bitstring, in_len, out_len*) inputs a *bitstring* of *in_len* bits, returning a string consisting of the leftmost *out_len* bits of *bitstring*. If *in_len* < *out_len*, the *bitstring* is padded on the right with (*out_len* - *in_len*) zeroes, and the result is returned.

c. *x*(*A*) is the *x*-coordinate of the point *A* on the curve, given in affine coordinates. An implementation may choose to represent points internally using other coordinate systems; for instance, when efficiency is a primary concern. In this case, a point **shall** be translated back to affine coordinates before *x*() is applied.

d. φ (*x*) maps field elements to non-negative integers, taking the bit vector representation of a field element and interpreting it as the binary expansion of an integer.

The precise definition of φ(*x*) used in steps 6 and 7 of the generate process

below depends on the field representation of the curve points. In keeping with the convention of FIPS 186-2, the following elements will be associated with each other (note that $m = seedlen$):

B: $| c_{m-1} | c_{m-2} | \ldots | c_1 | c_0 |$, a bitstring, with c_{m-1} being leftmost

Z: $c_{m-1}2^{m-1} + \ldots + c_2 2^2 + c_1 2^1 + c_0 \in Z$;

Fa: $c_{m-1}2^{m-1} + \ldots + c_2 2^2 + c_1 2^1 + c_0 \bmod p \in \mathrm{GF}(p)$;

Thus, any field element x of the form Fa will be converted to the integer Z or bitstring B, and vice versa, as appropriate.

e. * is the symbol representing scalar multiplication of a point on the curve.

The following process or its equivalent **shall** be used to generate pseudorandom bits (see step 8 of the generate process in Section 9.3):

Input:

1. *working_state*: The current values for *s, seedlen, p, a, b, n, P, Q,* and ~~*r_old* and~~ *reseed_counter* (see Section 10.3.1.1).
2. *requested_number_of_bits*: The number of pseudorandom bits to be returned to the generate function.
3. *additional_input*: The additional input string received from the consuming application.

Output:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS,** ~~**ERROR**~~ or an indication that a reseed is required before the requested pseudorandom bits can be generated.
2. *returned_bits*: The pseudorandom bits to be returned to the generate function.
3. *s*: The new value for the secret parameter in the *new_working_state*.
4. ~~*r_old*: The last output block.~~

~~5.~~ *block_counter*: The updated block counter for reseeding.

Dual_EC_DRBG Generate Process:

Comment: Check whether a reseed is required.

1. If $\left(block\_counter + \left\lceil \frac{requested\_number\_of\_bits}{outlen} \right\rceil \right) > reseed\_interval$, then return an indication that a reseed is required.

Comment: If *additional_input* is *Null*, set to *seedlen* zeroes; otherwise, **Hash_df** to

seedlen bits.

2. If (*additional_input_string* = *Null*), then *additional_input* = 0
   Else *additional_input* = **Hash_df** (**pad8** (*additional_input_string*), *seedlen*).
   Comment: Produce *requested_no_of_bits*, *outlen* bits at a time:
3. *temp* = the *Null* string.
4. *i* = 0.
5. *t* = *s* ⊕ *additional_input.* Comment: *t* is to be interpreted as a *seedlen*-bit unsigned integer. To be precise, *t* should be reduced mod *n*; the operation * will effect this.
6. *s* = φ(*x*(*t* * *P*)). Comment: *s* is a *seedlen*-bit number.
7. *r* = φ(*x*(*s* * *Q*)). Comment: *r* is a *seedlen*-bit number.
8. ~~If (*r* = *r_old*), then return an **ERROR**.~~
9. ~~*r_old* = *r*.~~
10. ~~*temp* = *temp* || (**rightmost** *outlen* bits of *r*).~~
11. 9. *additional_input*=0 Comment: *seedlen* zeroes; *additional_input_string* is added only on the first iteration.
12. 10. *block_counter* = *block_counter* + 1.
13. 11. *i* = *i* + 1.
14. 12. If (**len** (*temp*) < *requested_number_of_bits*), then go to step 5.
15. 13 *returned_bits* = **Truncate** (*temp*, *i* × *outlen*, *requested_number_of_bits*).
16. 14. Return **SUCCESS**, *returned_bits,* and *s*, ~~*r_old* and~~ *block_counter* for the *new_working_state*.

10.4 Auxilliary Functions

Derivation functions are internal functions that are used during DRBG instantiation and reseeding to either derive internal state values or to distribute entropy throughout a bitstring. Two methods are provided. One method is based on hash functions (see Section 10.4.1), and the other method is based on block cipher algorithms (see 10.4.2). The block cipher derivation function uses a **Block_Cipher_Hash** function that is specified in Section

10.4.3.

10.4.1 Derivation Function Using a Hash Function (Hash_df)

This derivation function is used by the **Hash_DRBG** and **Dual_EC_DRBG** specified Section 10.1.1 and 10.3.1, respectively. The hash-based derivation function hashes an input string and returns the requested number of bits. Let **Hash (...)** be the hash function used by the DRBG, and let *outlen* be its output length.

The following or an equivalent process **shall** be used to derive the requested number of bits.

Input:

1. *input_string*: The string to be hashed.
2. *no_of_bits_to_return*: The number of bits to be returned by **Hash_df.** The maximum length (*max_number_of_bits*) is implementation dependent, but **shall** be less than or equal to (255 × *outlen*). *no_of_bits_to_return* is represented as a 32-bit integer.

Output:

1. *status*: The status returned from **Hash_df.** The status will indicate **SUCCESS** or **ERROR_FLAG.**
2. *requested_bits* : The result of performing the **Hash_df.**

Hash_df Process:

1. If *no_of_bits_to_return* > *max_number_of_bits*, then return an **ERROR_FLAG.**
2. *temp* = the Null string.
3. $len = \left\lceil \frac{no\_of\_bits\_to\_return}{outlen} \right\rceil$.
4. *counter* = an 8-bit binary value representing the integer "1".
5. For *i* = 1 to *len* do

   Comment : In step 5.1, *no_of_bits_to_return* is used as a 32-bit string.

   5.1 *temp* = *temp* || **Hash** (*counter* || *no_of_bits_to_return* || *input_string*).

   5.2 *counter* = *counter* + 1.
6. *requested_bits* = Leftmost (*no_of_bits_to_return*) of *temp*.
7. Return **SUCCESS** and *requested_bits*.

10.4.2 Derivation Function Using a Block Cipher Algorithm (Block_Cipher_df)

This derivation function is used by the **CTR_DRBG** that is specified in Section 10.2. Let

Block_Cipher_Hash be the function specified in Section 10.4.3. Let *outlen* be its output block length, which is a multiple of 8 bits for the Approved block cipher algorithms, and let *keylen* be the key length.

The following or an equivalent process **shall** be used to derive the requested number of bits.

Input:

1. *input_string*: The string to be operated on. This string **shall** be a multiple of 8 bits.
2. *no_of_bits_to_return*: The number of bits to be returned by **Block_Cipher_df**. The maximum length (*max_number_of_bits*) is 512 bits for the currently approved block cipher algorithms.

Output:

1. *status*: The status returned from **Block_Cipher_df**. The status will indicate **SUCCESS** or **ERROR_FLAG**.
2. *requested_bits* : The result of performing the **Block_Cipher_df**.

Block_Cipher_df Process:

1. If (*number_of_bits_to_return* > *max_number_of_bits*), then return an **ERROR_FLAG**.
2. L = **len** (*input_string*)/8. Comment: L is the bitstring represention of the integer resulting from **len** (*input_string*)/8. L **shall** be represented as a 32-bit integer.
3. N = *number_of_bits_to_return*/8. Comment : N is the bitstring represention of the integer resulting from *number_of_bits_to_return*/8. N **shall** be represented as a 32-bit integer.

   Comment: Prepend the string length and the requested length of the output to the *input_string*.
3. $S = L \| N \| input\_string \| 0x80$.

   Comment : Pad S with zeros, if necessary.
4. While (**len** (S) mod *outlen*) $\neq 0$, $S = S \| 0x00$.

   Comment : Compute the starting value.
5. *temp* = the *Null* string.
6. $i = 0$. Comment : i **shall** be represented as a 32-bit integer, i.e., **len** (i) = 32.
7. K = Leftmost *keylen* bits of 0x00010203...1F.

8. While **len** (*temp*) < *keylen* + *outlen*, do

   8.1 $IV = i \| 0^{outlen - \text{len}(i)}$. Comment: The 32-bit integer represenation of *i* is padded with zeros to *outlen* bits.

   8.2 *temp* = *temp* || **Block_Cipher_Hash** (*K*, (*IV* || *S*)).

   8.3 *i* = *i* + 1.

   Comment: Compute the requested number of bits.

9. *K* = Leftmost *keylen* bits of *temp*.

10. *X* = Next *outlen* bits of *temp*.

11. *temp* = the *Null* string.

12. While **len** (*temp*) < *number_of_bits_to_return*, do

    12.1 *X* = **Block_Encrypt** (*K*, *X*).

    12.2 *temp* = *temp* || *X*.

13. *requested_bits* = Leftmost *number_of_bits_to_return* of *temp*.

14. Return **SUCCESS** and *requested_bits*.

10.4.3 Block_Cipher_Hash Function

The **Block_Encrypt** pseudo-function is used for convenience in the specification of the **Block_Cipher_Hash** function. This function is not specifically defined in this Recommendation, but has the following meaning:

Block_Encrypt: A basic encryption operation that uses the selected block cipher algorithm. The function call is:

output_block = **Block_Encrypt** (*Key*, *input_block*)

For TDEA, the basic encryption operation is called the forward cipher operation (see SP 800-67); for AES, the basic encryption operation is called the cipher operation (see FIPS 197). The basic encryption operation is equivalent to an encryption operation on a single block of data using the ECB mode.

For the **Block_Cipher_Hash** function, let *outlen* be the length of the output block of the block cipher algorithm to be used.

The following or an equivalent process **shall** be used to derive the requested number of bits.

Input:

1. *Key*: The key to be used for the block cipher opeation.
2. *data_to_hash*: The data to be operated upon. Note that the length of *data_to_hash* must be a multiple of *outlen*. This is guaranteed by steps 4 and 8.1 in Section

10.4.2.

Output:

1. *output_block*: The result to be returned from the **Block_Cipher_Hash** operation.

Block_Cipher_Hash Process:

1. *chaining_value* = 0^{outlen}. Comment: Set the first chaining value to *outlen* zeros.
2. *n* = **len** (*data_to_hash*)/*outlen*.
3. Split the *data_to_hash* into *n* blocks of *outlen* bits each forming $block_1$ to $block_n$.
4. For *i* = 1 to *n* do
   4.1 *input_block* = *chaining_value* ⊕ $block_i$.
   4.2 *chaining_value* = **Block_Encrypt** (*Key*, *input_block*).
5. *output_block* = chaining_value.
6. Return *output_block*.

11 Assurance

A user of a DRBG for cryptographic purposes requires assurance that the generator actually produces random and unpredictable bits. The user needs assurance that the design of the generator, its implementation and its use to support cryptographic services are adequate to protect the user's information. In addition, the user requires assurance that the generator continues to operate correctly. The assurance strategy for the DRBGs in this Recommendaion is depicted in Figure 15.

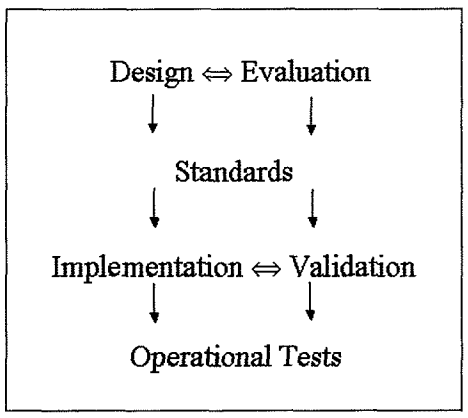

Figure 15: DRBG Assurance Strategy

The design of each DRBG in this Recommendation has received an evaluation of its security properties prior to its selection for inclusion in this Recommendation.

An implementation **shall** be validated for conformance to this Recommendation by a NVLAP accredited laboratory (see Section 11.2). The consuming application or cryptographic service that uses a DRBG **should** also be validated and periodically tested for continued correct operation. However, this level of testing is outside the scope of this Recommendation. Such validations provide a higher level of assurance that the DRBG is correctly implemented. Validation testing for DRBG processes consists of testing whether or not the DRBG process produces the expected result, given a specific set of input parameters (e.g., entropy input).

~~Operational (i.e., h~~Health~~)~~ tests on the DRBG **shall** be implemented within a DRBG boundary or sub-boundary in order to determine that the process continues to operate as designed and implemented. See Section 11.3 for further information.

A cryptographic module containing a DRBG **shall** be validated (see FIPS 140-2). The consuming application or cryptographic service that uses a DRBG ***should*** also be validated and periodically tested for continued correct operation. However, this level of testing is outside the scope of this Recommendation.

Note that any entropy input used for testing (either for validation testing or ~~operational/~~health testing) may be publicly known. Therefore, entropy input used for testing **shall not** knowingly be used for normal operational use.

11.1 Minimal Documentation Requirements

This Recommendation requires the development of a set of documentation that will provide assurance to users and (optionally) validators that the DRBGs in this

Recommendation have been implemented properly. Much of this documentation may be placed in a user's manual. This documentation **shall** consist of the following as a minimum:

- Document the method for obtaining entropy input.
- Document how the implementation has been designed to permit implementation validation and ~~operational~~ health testing.
- Document the type of DRBG (e.g., CTR_DRBG, Dual_EC_DRBG), and the cryptographic primitives used (e.g., AES-128, SHA-256).
- Document the security strengths supported by the implementation.
- Document features supported by the implemention (e.g., prediction resistance, the available elliptic curves, etc.).
- In the case of the CTR_DRBG, indicate whether a derivation function is provided. If a derivation function is not used, documentation **shall** clearly indicate that the implementation can only be used when full entropy input is available.
- Document any support functions other than ~~operational~~ health testing.

11.2 Implementation Validation Testing

A DRBG process **shall** be tested for conformance to this Recommendation. A DRBG **shall** be designed to be tested to ensure that the product is correctly implemented. A testing interface **shall** be available for this purpose in order to allow the insertion of input and the extraction of output for testing.

Implementations to be validated **shall** include the following:

- Documentation specified in Section 11.1.
- Any documentation or results required in derived test requirements.

11.3 ~~Operational/~~Health Testing

11.3.1 Overview

A DRBG implementation **shall** perform self-tests to ensure that the DRBG continues to function properly. Self-tests of the DRBG processes **shall** be performed as specified in Section 9.~~6~~5. A DRBG implementation may optionally perform other self-tests for DRBG functionality in addition to the tests specified in this Recommendation.

All data output from the DRBG boundary **shall** be inhibited while these tests are performed. The results from known-answer-tests (see Section 11.3.2) **shall not** be output as random bits during normal operation.

When a DRBG fails a self-test, the DRBG **shall** enter an error state and output an error indicator. The DRBG **shall not** perform any DRBG operations while in the error state, and

no pseudorandom bits **shall** be output when an error state exists. When in an error state, user intervention (e.g., power cycling, restart of the DRBG) **shall** be required to exit the error state (see Section 9.66).

11.3.2 Known Answer Testing

Known-answer testing **shall** be conducted as specified in Section 9.65. A known-answer test involves operating the DRBG with data for which the correct output is already known and determining if the calculated output equals the expected output (the known answer). The test fails if the calculated output does not equal the known answer. In this case, the DRBG **shall** enter an error state and output an error indicator (see Section 9.76).

The generalized known-answer testing is specified in Section 9.65. Testing **shall** be performed on all DRBG functions implemented.

Appendix A: (Normative) Application-Specific Constants

A.1 Constants for the Dual_EC_DRBG

The **Dual_EC_DRBG** requires the specifications of an elliptic curve and two points on the elliptic curve. One of the following NIST approved curves and points **shall** be used in applications requiring certification under FIPS 140-2. More details about these curves may be found in FIPS PUB 186-3, the Digital Signature Standard.

Each of following curves is given by the equation:

$$y^2 = x^3 - 3x + b \pmod{p}$$

Notation:

p - Order of the field F_p, given in decimal

r - order of the Elliptic Curve Group, in decimal . Note that r is used here for consistency with FIPS 186-3 but is referred to as n in the description of the **Dual_EC_DRBG (...)**

a – (-3) in the above equation

b - coefficient above

The x and y coordinates of the base point, ie generator G, are the same as for the point P.

~~A.1.1 Curve P-224~~

~~p = 26959946667150639794667015087019630673557916\~

~~26002630814351006629888l~~

~~r = 26959946667150639794667015087019625940457807\~

~~7144243917216827223680611~~

~~b = b4050a85 0c04b3ab f5413256 5044b0b7 d7bfd8ba 270b3943~~

~~2355ffb4~~

~~Px = b70e0cbd 6bb4bf7f 321390b9 4a03c1d3 56c21122 343280d6~~
~~115c1d21~~

~~Py = bd376388 b5f723fb 4c22dfe6 cd4375a0 5a074764 44d58199~~
~~85007e34~~

~~Qx = 68623591 6c11adfa f080a451 477fa27a f21248bc 916d3458~~
~~a583a3c9~~

~~Qy = 6060018a 24b35bc6 caccf3f0 7f2c6b43 4c47479c 55362c8f~~

~~5707adca~~

A.1.12 Curve P-256

p = 115792089210356248762697446949407573530086l4\
341529031419553363130886709785395l

r = 115792089210356248762697446949407573529996955\
224135760342422259061068512044369

b = 5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f6 3bce3c3e 27d2604b

Px = 6b17d1f2 e12c4247 f8bce6e5 63a440f2 77037d81 2deb33a0 f4a13945 d898c296

Py = 4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16 2bce3357 6b315ece cbb64068 37bf51f5

Qx = c97445f4 5cdef9f0 d3e05e1e 585fc297 235b82b5 be8ff3ef ca67c598 52018192

Qy = b28ef557 ba31dfcb dd21ac46 e2a91e3c 304f44cb 87058ada 2cb81515 1e610046

A.1.23 Curve P-384

p = 39402006196394479212279040100143613805079739\
270465446667948293404245721771496870329047266\
6088258938001861606973112319

r = 39402006196394479212279040100143613805079739\
270465446667946905279627659399113263569398956\
308152294913554433653942643

b = b3312fa7 e23ee7e4 988e056b e3f82d19 181d9c6e fe814112 0314088f 5013875a c656398d 8a2ed19d 2a85c8ed d3ec2aef

Px = aa87ca22 be8b0537 8eb1c71e f320ad74 6e1d3b62 8ba79b98 59f741e0 82542a38 5502f25d bf55296c 3a545e38 72760ab7

Py = 3617de4a 96262c6f 5d9e98bf 9292dc29 f8f41dbd 289a147c e9da3113 b5f0b8c0 0a60b1ce 1d7e819d 7a431d7c 90ea0e5f

Qx = 8e722de3 125bddb0 5580164b fe20b8b4 32216a62 926c5750 2ceede31 c47816ed d1e89769 124179d0 b6951064 28815065

Qy = 023b1660 dd701d08 39fd45ee c36f9ee7 b32e13b3 15dc0261 0aa1b636 e346df67 1f790f84 c5e09b05 674dbb7e 45c803dd

A.1.34 Curve P-521

p = 6864797660130609714981900799081393217269435 3\
0014330540939446345918554318339765605212255 9\
6406614545549772963113914808580371219879997 1\
6643812574028291115057151

r = 6864797660130609714981900799081393217269435 3\
0014330540939446345918554318339765539424505 7\
7463332171975329639963713633211138647686124 4\
0380340372808927070054 49

b = 051953eb 9618e1c9 a1f929a2 1a0b6854 0eea2da7 25b99b31 5f3b8b48
9918ef10 9e156193 951ec7e9 37b1652c 0bd3bb1b f073573d f883d2c3
4f1ef451 fd46b503 f00

Px = c6858e06 b70404e9 cd9e3ecb 662395b4 429c6481 39053fb5
21f828af 606b4d3d baa14b5e 77efe759 28fe1dc1 27a2ffa8
de3348b3 c1856a42 9bf97e7e 31c2e5bd 66

Py = 11839296 a789a3bc 0045c8a5 fb42c7d1 bd998f54 449579b4
46817afb d17273e6 62c97ee7 2995ef42 640c550b 9013fad0
761353c7 086a272c 24088be9 4769fd16 650

Qx = 1b9fa3e5 18d683c6 b6576369 4ac8efba ec6fab44 f2276171
a4272650 7dd08add 4c3b3f4c 1ebc5b12 22ddba07 7f722943
b24c3edf a0f85fe2 4d0c8c01 591f0be6 f63

Qy = 1f3bdba5 85295d9a 1110d1df 1f9430ef 8442c501 8976ff34
37ef91b8 1dc0b813 2c8d5c39 c32d0e00 4a3092b7 d327c0e7
a4d26d2c 7b69b58f 90666529 11e45777 9de

A.2 Using Alternative Points in the Dual_EC_DRBG()

The security of **Dual_EC_DRBG( )** requires that the points P and Q be properly generated. To avoid using potentially weak points, the points specified in Appendix A.1 **should** be used. However, an implementation may use different pairs of points, provided that they are *verifiably random*, as evidenced by the use of the procedure specified in Appendix A.2.1 below, and the self-test procedure in Appendix A.2.2. An implementation that uses alternative points generated by this Approved method **shall** have them "hard-wired" into its source code, or hardware, as appropriate, and loaded into the *working_state* at instantiation. To conform to this Recommendation, alternatively generated points **shall** use the procedure given in Appendix A.2.1, and verify their generation using Appendix A.2.2.

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 point *P* **shall** remain the generator point *G* given in Appendix A.1 for the selected curve. (This minor restriction simplifies the test procedure to verify just one point each time.)

The point *Q* **shall** be generated using the procedure specified in ANS X9.62. The following input is required:

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

If the output from the ANS X9.62 generation procedure is "failure", a different *SEED* must be used.

Otherwise, the output point **shall** be used as the point *Q*.

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

To insure that the point *Q* has been generated appropriately, an additional self-test procedure **shall** be performed whenever the instantiate function is invoked. Section 9.5~~6~~.1~~2~~ specifies that known-answer tests on the instantiate function be performed prior to creating an operational instantiation. As part of those tests, an implementation of the generation procedure in ANS X9.62 **shall** be called with the *SEED* value used to generate the alternate *Q*. The point returned **shall** be compared with the stored value of *Q* used in place of the default value (see Appendix A.1 of this Recommendation). If the generated value does not match the stored value, the implementation **shall** halt with an error condition.

Appendix B : (Normative) Conversion and Auxilliary Routines

B.1 Bitstring to an Integer

Input:

1. $b_1, b_2, ..., b_n$ The bitstring to be converted.

Output:

1. x The requested integer representation of the bitstring.

Process:

1. Let $(b_1, b_2, ..., b_n)$ be the bits of b from leftmost to rightmost.
2. $x = \sum_{i=1}^{n} 2^{(n-i)} b_i$.
3. Return x.

In this Recommendation, the binary length of an integer x is defined as the smallest integer n satisfying $x < 2^n$.

B.2 Integer to a Bitstring

Input:

1. x The non-negative integer to be converted.

Output:

1. $b_1, b_2, ..., b_n$ The bitstring representation of the integer x.

Process:

1. Let $(b_1, b_2, ..., b_n)$ represent the bitstring, where $b_1 = 0$ or 1, and b_1 is the most significant bit, while b_n is the least significant bit.
2. For any integer n that satisfies $x < 2^n$, the bits b_i **shall** satisfy:

$$x = \sum_{i=1}^{n} 2^{(n-i)} b_i .$$

3. Return $b_1, b_2, ..., b_n$.

In this Recommendation, the binary length of the integer x is defined as the smallest integer n that satisfies $x < 2^n$.

B.3 Integer to an Octet String

Input:

1. A non-negative integer x, and the intended length n of the octet string satisfying

$$2^{8n} > x.$$

Output:

1. An octet string O of length n octets.

Process:

1. Let $O_1, O_2, ..., O_n$ be the octets of O from leftmost to rightmost.
2. The octets of O **shall** satisfy:

$$x = \Sigma\, 2^{8(n-i)} O_i$$

for $i = 1$ *to* n.

3. Return O.

B.4 Octet String to an Integer

Input:

1. An octet string O of length n octets.

Output:

1. A non-negative integer x.

Process:

1. Let $O_1, O_2, ..., O_n$ be the octets of O from leftmost to rightmost.
2. x is defined as follows:

$$x = \Sigma\, 2^{8(n-i)} O_i$$

for $i = 1$ *to* n.

3. Return x.

B.5 Converting Random Numbers from/to Random Bits

The random values required for cryptographic applications are generally of two types: either a random bitstring of a specified length, or a random integer in a specified interval. In some cases, a DRBG may return a random number in a specified interval that needs to be converted into random bits; in other cases, a DRBG returns a random bitstring that needs to be converted to a random number in a specific range.

B.5.1 Converting Random Bits into a Random Number

In some cryptographic applications sequences of random numbers are required (a_0, a_1, a_2,...) where:

i) Each integer a_i satisfies $0 \le a_i \le r\text{-}1$, for some positive integer r (the *range* of the

random numbers);

ii) The equation $a_i = s$ holds, with probability almost exactly $1/r$, for any $i \geq 0$ and for any s ($0 \leq s \leq r$-1);

iii) Each value a_i is statistically independent of any set of values a_j ($j \neq i$).

Four techniques are specified for generating sequences of random numbers from sequences of random bits.

If the range of the number required is $a \leq a_i \leq b$ rather than $0 \leq a_i \leq r$-1, then a random number in the desired range can be obtained by computing $a_i + a$, where a_i is a random number in the range $0 \leq a_i \leq b$-a (that is, when $r = b$-a+1).

B.5.1.1 The Simple Discard Method

Let m be the number of bits needed to represent the value (r–1). The following method may be used to generate the random number a:

1. Use the random bit generator to generate a sequence of m random bits, (b_0, b_1, …, b_{m-1}).
2. Let $c = \sum_{i=0}^{m-1} 2^i b_i$.
3. If $c < r$ then put $a = c$, else discard c and go to Step 1.

This method produces a random number a with no skew (no bias). A possible disadvantage of this method, in general, is that the time needed to generate such a random a is not a fixed length of time because of the conditional loop.

The ratio $r/2^m$ is a measure of the efficiency of the technique, and this ratio will always satisfy $0.5 < r/2^m \leq 1$. If $r/2^m$ is close to 1, then the above method is simple and efficient. However, if $r/2^m$ is close to 0.5, then the simple discard method is less efficient, and the more complex method below is recommended.

B.5.1.2 The Complex Discard Method

Choose a small positive integer t (the number of same-size random number outputs desired), and then let m be the number of bits in (r^t–1). This method may be used to generate a sequence of t random numbers (a_0, a_1, … , a_{t-1}):

1. Use the random bit generator to generate a sequence of m random bits, (b_0, b_1, …, b_{m-1}).
2. Let $c = \sum_{i=0}^{m-1} 2^i b_i$.
3. If $c < r^t$, then

   let (a_0, a_1, …, a_{t-1}) be the unique sequence of values satisfying $0 \leq a_i \leq r$ -1 such

that $c = \sum_{i=0}^{t-1} r^i a_i$

else discard c and go to Step 1.

This method produces random numbers $(a_0, a_1, \ldots, a_{t-1})$ with no skew. A possible disadvantage of this method, in general, is that the time needed to generate these numbers is not a fixed length of time because of the conditional loop. The complex discard method is guaranteed to produce a sequence of random outputs for each iteration and, therefore, may have better overall performance than the simple discard method if many random numbers are needed.

The ratio $r^t/2^m$ is a measure of the efficiency of the technique, and this ratio will always satisfy $0.5 < r^t/2^m \leq 1$. Hence, given r, it is recommended to choose t so that t is the smallest value such that $r^t/2^m$ is close to 1. For example, if $r = 3$, then choosing $t = 3$ means that $m = 5$ (as r^t is 27) and $r^t/m = 27/32 \approx 0.84$, and choosing $t = 5$ means that $m = 8$ (as r^t is 243) and $r^t/m = 243/256 \approx 0.95$. The complex discard method coincides with the simple discard method when $t = 1$.

B.5.1.3 The Simple Modular Method

Let m be the number of bits needed to represent the value $(r-1)$, and let s be a security parameter. The following method may be used to generate one random number a:

1. Use the random bit generator to generate a sequence of $m+s$ random bits, $(b_0, b_1, \ldots, b_{m+s-1})$.
2. Let $c = \sum_{i=0}^{m+s-1} 2^i b_i$.
3. Let $a = c \bmod r$.

The simple modular method can be coded to take constant time. This method produces a random value with a negligible skew, that is, the probability that $a_i=w$ for any particular value of w $(0 \leq w \leq r\text{-}1)$ is not exactly $1/r$. However, for a large enough value of s, the difference between the probability that $a_i=w$ for any particular value of w and $1/r$ is negligible. The value of s **shall** be greater than or equal to 64. ~~[Why 64? What is the relationship between s and the security strength?]~~

B.5.1.4 The Complex Modular Method

Choose a small positive integer t (the number of same-size random number outputs desired) and a security parameter s; let m be the number of bits in $(r^t - 1)$. The following method may be used to generate a sequence of t random numbers $(a_0, a_1, \ldots, a_{t-1})$:

1. Use the random bit generator to generate a sequence of $m+st$ random bits, $(b_0, b_1, \ldots, b_{m+st-1})$.

2. Let $c = \sum_{i=0}^{m+s-1} 2^i b_i \mod r^l$.
3. Let $(a_0, a_1, \ldots, a_{t-1})$ be the unique sequence of values satisfying $0 \le a_i \le r-1$ such that $c = \sum_{i=0}^{t-1} r^i a_i$.

The complex modular method is guaranteed to produce a sequence of random outputs with each iteration and, therefore, may have better overall performance than the simple modular method if many random numbers are needed. This method produces a random value with a negligible skew; that is, the probability that $a_i=w$ for any particular value of w ($0 \le w \le r-1$) is not exactly $1/r$. However, for a large enough value of s, the difference between the probability that $a_i=w$ for any particular value of w and $1/r$ is negligible. The value of s **shall** be greater than or equal to 64. The complex modular method coincides with the simple modular method when $t=1$.

B.5.2 Converting a Random Number into Random Bits

B.5.2.1 The No Skew (Variable Length Extraction) Method

This is a method of extracting random unbiased bits from a random number modulo a number n. First, a toy example is provided in order to explain how the method works, and then pseudocode is given.

For the toy example, the insight is to look at the modulus n and the random number r as bits, from left to right, and to partition the possible values of r into disjoint sets based on the largest size of random bits that might be extracted. As a small example, if $n = 11$, then the binary representation of n is b'1011', and the possible values of r (in binary) are as follows:

0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010.

Let the leftmost bit be considered as the bit 4, and the rightmost bit be considered as the bit 1.

1. As the 4th bit of n is b'1', look at the 4th bit of r.
2. If the 4th bit of r is b'0', then the remaining 3 bits can be extracted as unbiased random bits. This forms a class of [0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111] and maps each respective element into the 3-bit sequences [000, 001, 010, 011, 100, 0101, 110, 111], each of which is unbiased, and the process is completed
3. If the 4th bit of r is b'1', then r falls into the remainder [1000, 1001, 1010], and the process needs to continue with step 4 in order to extract unbiased bits.
4. As the 3rd bit of n is b'0', the 3rd bit of r is always b'0' in the class determined in step 3; therefore the 3rd bit of r is already known to be biased, so the analysis moves to the next bit (step 5).

5. The 2nd bit of *n* is b'1', so this forms a subclass [1000, 1001], from which one random unbiased bit can be extracted, namely the 1st bit.

   The remaining value of 1010 cannot be used to extract random bits. However, obtaining this value is not usual. For this tiny example: 8/11 of the time, 3 unbiased random bits can be extracted; 2/11 of the time, 1 unbiased bit can be extracted; and 1/11, no unbiased bits can be extracted. As can be seen, it is not known ahead of time how many unbiased bits will be able to be extracted, although the average will be known.

Let both the modulus *n* and the random *r* values have *m* bits. This means that *n*(*m*) = b'1', although *r*(*m*) may be either b'1' or b'0'.

1. *outlen* = 0.
2. Do *i* = *m* to 1 by –1

   Comment: if *n*(*i*) = b'0', or *r*(*i*) = b'1', then this is a skew situation; the routine cannot extract *i*-1 unbiased bits, so the index is shifted right to check next bit

   2.1 If ((*n*(*i*) = b'0') or (*r*(*i*) = b'1')), then go to step 2.5.

   2.2 *outlen* = i-1.

   2.3 *output* = *r*(*outlen*,1).

   2.4 *i* = 1 Comment: all unbiased bits possible have been extracted, so exit .

   2.5 Continue

The extraction takes a variable amount of time, but this varying amount of time does not leak any information to a potential adversary that can be used to attack the method.

B.5.2.2 The Negligible Skew (Fixed Length Extraction) Method

A possible disadvantage of the No Skew (Variable Length Extraction) Method of Appendix B.5.2.1 is that it takes a variable amount of time to extract a variable number of random bits. To address this concern and to simplify the extraction method, the following method is specified that extracts a fixed length of random bits with a negligible skew. This method exploits the fact that the modulus *n* is known before the extraction occurs.

1. Examine the modulus considered as a binary number from left to right, and determine the index bit such that there are at least 16 b'1' bits to the left. Call this bit *i*.
2. Extract random bits from the random number *r* by truncating on the left up to bit *i*. This is the output = *r*(*i*,1).

This method is especially appropriate when the high order bits of the modulus are all set to

b'1' for efficiency reasons, as is the case with the NIST elliptic curves over prime fields.

This method is acceptable for elliptic curves, based on the following analysis. When considering the no skew method, once the random bits are extracted, it is obvious that less than the full number of random bits can be extracted, and the extraction result will still be random. The truncation of more bits than necessary is acceptable. What about truncation of too few bits? For a random number, the no skew extraction process would continue only if the 16 bits of *r* corresponding to the b'1' bits in *n* are all zero. For a random number, this occurs about once every 2^{16} times. As the modulus is at least 160 bits, this means that 144 bits with a skew are extracted in this case. On average, once every 9,437,184 output bits (or more), there will be a 144-bit substring somewhere in that total that has a skew, which will have the leftmost bit or bits tending to a binary zero bit or bits. This skew could be as little as one bit. However, an adversary will not know exactly where this skewed substring occurs. The 9,437,184 total output bits will still be overwhelmingly likely to be within the statistical variation of a random bitstring; that is, the statistical variation almost certainly will be much greater than this negligible skew.

Appendix C: (Normative) Entropy and Entropy Sources

An examination of the DRBG algorithms in this Recommendation reveals a common feature: each of them takes entropy input, produces a *seed* and applies an algorithm to produce a potentially large number of pseudo-random bits. The most important feature of the interaction between the entropy input and the algorithm is that if an adversary doesn't know the entropy input, then he can't tell the difference between the pseudo-random bits and a stream of truly random bits, let alone predict any of the pseudorandom bits. On the other hand, if he knows (or can guess) the entropy input, then he will be able to predict or reproduce the pseudorandom bits. ~~tell the difference between the pseudo-random bits and a stream of truly random bits. Conversely, if he knows (or can guess) the seed, then he can easily distinguish the output from random, since the algorithm is deterministic.~~ Thus, the security of the DRBG output is directly related to the adversary's inability to guess the entropy input.

C.1 What is Entropy ?

The word "entropy" is used to describe a measure of randomness, i.e., a description of how hard a value is to guess. Entropy is a measure of uncertainty or unpredictability and is dependent on the probabilities associated with the possible results for a given "event" (e.g., a throw of a die or flip of a coin).

In this Recommendation, entropy is relative to an adversary and his ability to observe or predict a value. If the adversary has no uncertainty about the value, then the entropy is zero (and so is the security of the consuming application that relies on the DRBG). Any assessment of the entropy of a particular value is actually an assessment of how much of the value is unknown to the adversary.

C.2 Entropy Source

Entropy is obtained from an entropy source. The entropy input required to seed or reseed a DRBG **shall** be obtained either directly or indirectly from an entropy source (see Appendix D for information on RBG construction). The entropy source is the critical component of an RBG that provides un-guessable values for the deterministic algorithm to use as entropy input for the random bit generation process.

Every entropy source must include some process that is unpredictable. An intuitive (although ~~often~~ usually impractical) example is tossing a coin and recording the sequence of heads and tails. More likely, the entropy source will be an electronic process, such as a noisy diode, which receives a constant input voltage level and outputs a continuous, normally distributed analog voltage level. Other possibilities include thermal noise or radioactive decay that are measured by appropriate instruments. The unpredictability could involve human interaction with an otherwise deterministic system, such as the sampling of a high-speed counter whenever a human operator presses a key on a keyboard. In any case, there ~~shall~~ be something happening that is unpredictable to an adversary, either fundamentally unpredictable (e.g., when the next particle is detected by a Geiger counter),

or unpredictable from a practical point of view (e.g., the adversary won't know the exact value of a high-speed counter if he isn't close enough to the human pressing a key).

Figure C-1 provides a generic model for an entropy source. A noise source (e.g., a noisy diode or a coin flip) provides the entropy, which is then converted to bits (i.e., digitized). ~~Collecting entropy from an entropy source requires obtaining numerous samples, where each sample is the result from a given type of event). Once sufficient samples have been gathered, they generally need to be converted to bits (e.g., an analog voltage will be mapped to some digital value, or coin tosses could be mapped to ones and zeros).~~ Some entropy sources will perform further processing (conditioning) on the resulting bits, guaranteeing unbiased output. An entropy source may process the bits to the point where the output bitstring will have full entropy; i.e. the entropy of the bitstring will be (nearly) the same as its length. In this case, the entropy source will usually include a conditioning routine, and the entropy source is often called a *conditioned entropy source*.

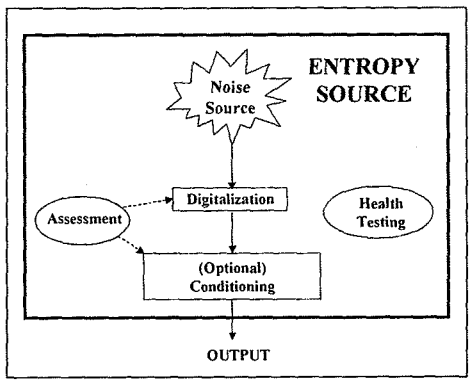

Figure C-1: Entropy Source Model

An assessment is made of the amount of entropy that has been obtained. Typically, this assessment is performed directly on the digitized data, although it may be ~~perfomed, perhaps on the digitalized data or~~ on the data resulting from the conditioning process (see Appendix C.~~2~~3). Health tests are performed to determine that the entropy source is performing correctly.

Before ~~a source of entropy~~an entropy source is selected for providing entropy input to a DRBG, a thorough evaluation of the amount of entropy it is capable of providing **shall** be ~~determined~~performed. ~~When a suitable entropy source is selected, a further assessment may be required to ensure that adequate entropy is actually provided when required by the DRBG (see Appendix C.2).~~

Guidance on the selection and use of entropy sources is currently under development and is expected to be provided as a NIST Recommendation in the future.

C.~~2~~3 Entropy Assessment

A DRBG requires a predetermined amount of entropy in the entropy input that is used to seed or reseed an instantiation in order to provide the requested DRBG security strength. Therefore, the amount of actual entropy obtained from an entropy source **shall** be assessed before providing it as entropy input. A~~;~~ ~~a~~ means of measuring the entropy is required. Note that the actual entropy provided in a given string of entropy input bits is less than or equal

to the length of that bitstring; i.e., each bit of the entropy input has (at most) one bit of entropy; multiple bits of the entropy input may be required to provide one bit of entropy.

There are many entropy measures defined in information theory; this Recommendation uses a very conservative measure that is known as *min-entropy* (H_{min})~~, and is defined as~~. Suppose that the digitized Noise Source produces one of n possible outputs at each sampling, with the i^{th} possible outcome having a probability of p_i. The min-entropy of the outputs is:

$$H_{min} = -\lg_2(p_{max})$$

where ~~each possible value is p_i, and~~ p_{max} is the maximum probability of the p_i. H_{min} is expressed in bits~~ and is the amount of entropy that is expected in each event that produces a value of p_i~~. Another, more commonly used measure of entropy is Shannon entropy. However, min-entropy is a more conservative estimate of entropy than Shannon entropy, since min-entropy is always less than Shannon entropy. Therefore, the more conservative estimate is used in this Recommendation

For example, suppose that a noisy diode is used as a source of entropy, and that the diode has possible voltages divided into 16 intervals (column 1), with each interval assigned a 4-bit string value from 0000 to 1111 (column 2). Whenever the diode is sampled, the result is digitized and converted to the 4-bit value indicagted in column 2. The probability of each interval has been determined for this diode and is provided in column 3. Note that other diodes may behave differently.

Collecting entropy from an entropy source requires obtaining numerous samples, where each sample is the result from a given type of event~~)~~. Once sufficient samples have been gathered, they generally need to be converted to bits (e.g. an analog voltage will be mapped to some digital value, or coin tosses could be mapped to ones and zeros).

Table C-1 : Voltages Digitaization Ranges and Probabilities

Sampled Voltage	Digitized Output	Probability (p_i)
$-\infty < Z < 2.5$	0000	0.000233
$2.5 \le Z < 3$	0001	0.001117
$3 \le Z < 3.5$	0010	0.004860
$3.5 \le Z < 4$	0011	0.016540

Sampled Voltage	Digitized Output	Probability (p_i)
$4 \le Z < 4.5$	0100	0.044057
$4.5 \le Z < 5$	0101	0.091848
$5 \le Z < 5.5$	0110	0.149882
$5.5 \le Z < 6$	0111	0.191462
$6 \le Z < 6.5$	1000	0.191462
$6.5 \le Z < 7$	1001	0.149882
$7 \le Z < 7.5$	1010	0.091848
$7.5 \le Z < 8$	1011	0.044057
$8 \le Z < 8.5$	1100	0.016540
$8.5 \le Z < 9$	1101	0.004860
$9 \le Z < 9.5$	1110	0.001117
$9.5 \le Z < \infty$	1111	0.000233

For this diode, the most likely digitized outputs are 0111 and 1000, each with a probability of 0.191462. Therefore, $p_{max} = 0.191462$. Using the min-entropy formula above:

$$H_{min} = -\lg_2(p_{max}) = -\lg_2(0.19462) = 2.38487.$$

This means that for each 4-bit sample from this diode, an entropy of 2.38487 bits is expected.

One useful fact about min-entropy is that if two samples are independent (e.g., samplings of the same noisy diode), then the entropy of their concatenation is the sum of their entropy. This makes sense; if the samples are independent, then guessing one sample provides no information for guessing another one. If various events are concatenated, then the min-entropy for each event is added to find the min-entropy of the concatenated events. In the noisy diode example, if ~~the~~a sample has a min-entropy of 2.38487 bits, then ten

samples taken together have a min-entropy of 23.8487 bits, and one hundred samples have a min-entropy of 238.487 bits.

These entropy measures relate directly to the security strengths of the Approved DRBG algorithms. When the entropy source is used to provide entropy input for a DRBG, each sample will provide a bitstring, along with the assessed amount of entropy in that bitstring. If a single sample does not provide sufficient entropy for the DRBG, a sequence of bitstrings are obtained and concatenated with each other until the sum of the entropy assessments for the samples is equal to or greater than the entropy required by the DRBG. For example, to provide entropy input that is appropriate to instantiate a DRBG with a security strength of 128 bits, at least 54 samplings of the diode are required ($128/2.38487 = 53.67 \approx 54$) and would result in a bitstring of 216 bits to provide at least 128 bits of entropy. ~~Note that the samplings could actually contain more or less entropy than expected.more entropy, because the min-entropy is the lower bound. The samplings may also contain less entropy if the assessment of min-entropy is in error, or if the distribution of the entropy from the noise source has changed (i.e., the noise source is behaving differently than expected).~~

C.4 Coin Flipping Entropy Source Example

Coin flipping (sometimes called coin tossing) is perhaps the most straightforward example of an entropy source, although it may be impractical to actually use in many cases. However, for the occasional seeding of a DRBG when other entropy sources are not available, coin flipping may be appropriate. This Recommendation allows the generation of random bits as the entropy input for a DRBG using this coin-flipping process when strict procedures are used to enforce accurate use of the process and protect the secrecy of the results.

The coin flipping procedure described here may be used as an entropy source because of the indepence of the coin flips. The procedure is as follows:

1. Select a single coin to be used for the procedure.
2. Determine the entropy requirement (x) for the DRBG to be instantiated.
3. Flip the coin until at least x heads and x tails have appeared, recording each coin flip result in order. Note that there will be at least 256 coin flips, and possibly several more.
4. Convert each head to either a zero or a one; convert each tail to the other value.
5. The entire string of zeroes and ones **shall** be used as the entropy input.

Appendix D: (Normative) Constructing a Random Bit Generator (RBG) from Entropy Sources and DRBG Mechanisms

This Recommendation is primarily concerned with the DRBG algorithms for generating pseudorandom outputs and how they are to be implemented. Some discussion of entropy sources that may be used to provide entropy input are provided in Appendix C. This appendix briefly describes how to combine the entropy source with a DRBG mechanism to create an Approved RBG.

D.1 Entropy Input for a DRBG

Section 8.6.5 states that the source of a DRBG's entropy input may be 1) an Approved Non-deterministic Random Bit Generator (NRBG), 2) an Approved DRBG (or chain of Approved DRBGs) or 3) an Approved entropy source. A clarification of concepts may be helpful at this point.

a. An NRBG contains an entropy source (see Appendix C.1) and performs algorithmic processing on the entropy source output in order to produce an output with full entropy (see Figure D-1).

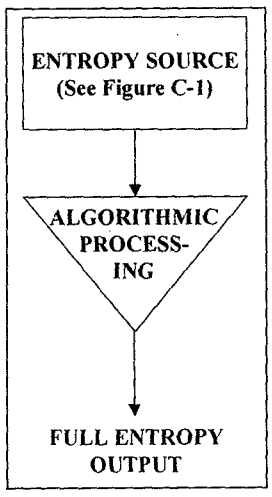

Figure D-1: NRBG

b. A DRBG is defined in the body of this Recommendation. To form a chain of DRBGs (see the chain of two DRBGs in Figure D-2), the entropy input for the instantiation of the first DRBG (the highest DRBG in the chain) **shall** be obtained from a "true" source of entropy (i.e., an Approved NRBG or an Approved entropy source). Each subordinate DRBG is instantiated with entropy input acquired from an entropy request to a higher DRBG in the chain; the entropy input **shall** contain sufficient entropy to support the requested security strength for the subordinate DRBG. The security strength provided by the higher level DRBG **shall** be equal to or greater than the security strength of any subordinate DRBG.

~~a~~c. An entropy source provides entropy source output (see Appendix C.1). This entropy source output may be used as the entropy input for a DRBG; i.e., the entropy input source may be the output of an entropy source (see DRBG A and the entropy input from the entropy source in Figure ~~??~~D-2).

~~b. An NRBG contains an entropy source (see Appendix C.1) and performs algorithmic processing on the entropy source output in order to produce an output with full entropy (see Figure ??).~~

~~c. A DRBG is defined in the body of this Recommendation.~~

~~d. To form a chain of DRBGs (see Figure ??), the entropy input for the first DRBG (the highest DRBG in the chain) **shall** be obtained from a "true" source of entropy (i.e., an Approved NRBG or an Approved entropy source whose entropy characteristics are known). The entropy **shall** be equal to or greater than the entropy required by any of the subordinate DRBGs.~~

~~Each subordinate DRBG is instantiated with entropy input acquired from an entropy request to a higher DRBG in the chain. The entropy input **shall** contain sufficient entropy to support the requested security strength.~~

~~e.~~An Approved entropy source by itself (i.e., not part of an NRBG) may or may not provide full entropy. However, the entropy resource will provide an assessment of the amount of entropy available in the output.

A (complete) RBG that incorporates a DRBG also includes the source of entropy input.

When designing such an RBG ~~using a DRBG~~, there are a number of concerns to be addressed in addition to the DRBG to be selected, including the entropy input source to be used, how readily the entropy input to the DRBG can be provided, and how the DRBG maintains its internal state information from one request to the next. Appendix G provides a discussion on DRBG selection, and Appendix C provides some basic discussion on entropy sources. This appendix includes discussions about using entropy input sources whose output may or may not be readily available and discusses internal state persistence.

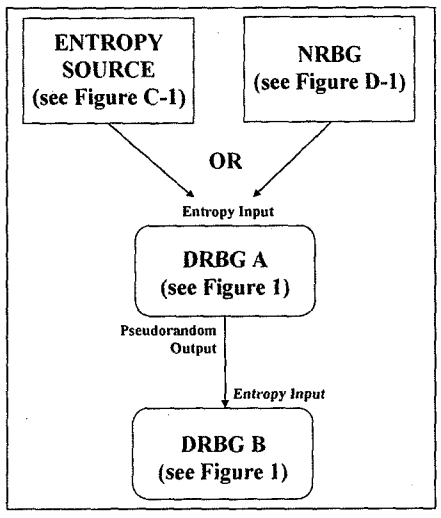

Figure D-2: Chain of DRBGs

D.~~1~~2 Availability of Entropy Input for a DRBG

~~The source of a DRBG's entropy input may be an Approved NRBG, an Approved DRBG or an Approved entropy source whose entropy characteristics are known (see Section 8.6.5).~~ The choice of an entropy input source ~~may~~ will determine the specific "features" that an RBG can offer a consuming application (e.g., whether reseeding or prediction resistance is practical). Whenever entropy inpu~~t~~ is requested by a DRBG, the entropy input source must provide sufficient entropy to support the security strength intended for the DRBG.~~.~~ The entropy input source may be able to provide entropy whenever requested (i.e., entropy is readily available on demand). On the other hand, the entropy input source may provide entropy too slowly to honor "frequent" requests (e.g., the entopy input source

may, in practice, be able to provide entropy only during instantiation). In any event, the entropy input must be provided to the DRBG mechanism via a secure (i.e., private and authentic) channel.

~~the entropy provided for instantiation must be provided over a secure (private and authentic) channel.~~

~~Over time, a DRBG may be able to accumulate additional entropy from inputs provided by the user or consuming application (i.e., *additional_input*). For this reason, the DRBG implementation **should** accept additional input whenever possible. Implementations that have values that may have some entropy, such as timestamps or nonces from protocol runs, **should** provide these values to the DRBG as additional inputs.~~

D.2.11 Using a Readily Available Entropy Input Source

The ideal situation for a DRBG is to have ready access to some entropy input source that provides entropy input (immediately) upon request. The entropy input source provides bitstrings, along with a promise about how much entropy is available.

When the DRBG has a readily available source of entropy input, reseeding and instantiation can be ~~done~~ performed on demand, requests for prediction resistance can be honored, and a DRBG can be reseeded when it has produced the maximum number of outputs (i.e., the *reseed_interval* is reached).

Upon each request for entropy input, the status of the request is returned to the calling function (i.e., the instantiate or reseed function). A failure of the entropy input source has the following consequences:

- If the failure of the entropy source is detected, the DRBG functions are designed to ~~indicate the error~~return an error status and enter the error state (see Section 9.~~7~~6). No further output is produced until the failure is corrected.
- If the failure is not immediately detected, the DRBG will continue to provide output, based on the entropy currently available in the internal state.

  If the failure occured prior to or during instantiation, an undetected failure would be catastrophic, as the DRBG would totally fail to provide the promised security strength. Therefore, extreme care must be taken to ensure that a DRBG is instantiated with sufficient entropy.

  If the failure occurred subsequent to instantiation, a request for prediction resistance would not result in prediction resistance being provided; however, the security strength of the output would be based on whatever entropy had previously been obtained.

  If the failure occured prior to or during a normal reseed (at the end of the *reseed_interval*), the security strength of the output would be based on whatever entropy had previously been obtained. If the implemented *reseed_interval* is the

maximum that can be supported by the DRBG (see the tables in Section 10), then the security provided by the DRBG algorithm is no longer assured. Therefore, the use of a *reseed_interval* that is significantly less than the maximum interval is recommended. This would provide additional time for the entropy source failure to be detected.

D.2.2 No Readily Available Entropy Input Source

Many implementations of DRBGs will not have ready access to an entropy input source; however, a. An implementation of this type has the following requirements:

The DRBG must be instantiated at a time when the DRBG actually has does have access to some reliable entropy input source. In some applications, the entropy input 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 internal state for as long as the DRBG may be called upon to generate outputs. This typically requires some kind of persistent memory (e.g., flash memory) or a means of securely saving the internal state to avoid losing the internal state during power down.

Over time, a DRBG may be able to accumulate additional entropy from inputs provided by the user or consuming application as *additional input*. For this reason, the DRBG implementation **should** accept additional input whenever possible. Implementations that have values that may have some entropy, such as timestamps or nonces from protocol runs, **should** provide these values to the DRBG as additional inputs.

D.33 Persistence ConsiderationsSaving the Internal State

A DRBG is provided with entropy input during instantiation, and the instantiation exists for as long as the internal state is maintained. In many environments, the internal state can be maintained for a very long time because power is continually available during that time, or the internal state is stored in persistent memory that is not affected by power fluctuations.

However, there are environments in which a DRBG does not have continual power, and the persistent memory may have limitations on the number of times that it can be changed without failing (e.g., flash memory is used to store the internal state); any DRBG whose internal state is saved in this limited access memory should be used conservatively, i.e., to provide outputs and changes to the internal state as infrequently as possible. If several random values are required whenever power is available, then a prudent design may be to instantiate a second DRBG from the DRBG using the limited access memory to store its internal state. The second DRBG's internal state may not reside in persistent memory. Let the DRBG using the limited access memory to store the internal state be called $DRBG_{source}$; let the second DRBG be called $DRBG_{short\text{-}lived}$.

$DRBG_{source}$ can be used to instantiate $DRBG_{short\text{-}lived}$ whenever power becomes available, and the $DRBG_{short\text{-}lived}$ instantiation only exists for as long as the power continues to be

available. The security strength of $DRBG_{short\text{-}lived}$ is dependent on the security strength provided by $DRBG_{source}$, which must have been instantiated when sufficient entropy was available as specified in Appendix D.1. An example of this case might be a smart card whose $DRBG_{source}$ is instantiated by the manufacturer or issuer, and the smart card only receives power thereafter when inserted into a smart card reader. This case is depicted in Figure D-2 by considering DRBG B to be $DRBG_{short\text{-}lived}$.

The following is a common method for interacting between the two DRBGs. The method is an adaptation of a concept that uses seed files in currently implemented RBGs.

1. Whenever power is available, a generate request is sent to $DRBG_{source}$. $DRBG_{source}$ generates the requested output and provides it to $DRBG_{short\text{-}lived}$ as entropy input for instantiation. The consuming application using $DRBG_{short\text{-}lived}$ may provide additional input to $DRBG_{short\text{-}lived}$ as a *personalization_string* during the instantiation process.

~~1~~2. After $DRBG_{short\text{-}lived}$ provides one or more outputs as requested by its consuming application, *k*-bits of additional output are generated by $DRBG_{short\text{-}lived}$, **where $k \geq 3/2$ *security_strength*.** [EBB2]**The *k*-bit output**, along with any other application data that might contain entropy, is**and** provided as *additional_input* to $DRBG_{source}$ in a generate request at some time before the power is removed. This will result in an update of the internal state of $DRBG_{source}$. Any resulting output from this request is ignored.

If $DRBG_{short\text{-}lived}$ generates a large number of outputs or persists for a long period of time, and it is unknown how long the power will be available, $DRBG_{source}$ **should periodically perform this process to ensure that** it is updated at least once before the power is removed.

~~D.4 Seeding Many DRBGs from One DRBG~~

~~Some applications may benefit from using different DRBGs for different applications. Each of these DRBGs must be seeded with sufficient entropy to support the DRBG's security strength. If an entropy source is not readily available for doing so, several DRBGs may be seeded from a single DRBG (the parent DRBG) or chain (e.g., hierarchy) of DRBGs. However, this must be done with some care, to avoid introducing new weaknesses.~~

- ~~The parent DRBG **shall** be instantiated with as much entropy as possible. The entropy **shall** be equal to or greater than the entropy required by any of the subordinate DRBGs.~~
- ~~Each subordinate DRBG is instantiated with entropy input acquired from an entropy request to the parent DRBG (or a higher DRBG in the chain). The entropy input **shall** contain sufficient entropy to support the requested security level.~~

Appendix E: (Informative) Security Considerations when

E.1 Extracting Bits in the Dual_EC_DRBG (...)

E.1.1 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 Recommendation, $k/2^{N-1}$ 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 Appendix E.2.

~~For the mod p curves (i.e, a *Prime field curve*), there is a potential bias in the output due to the modular arithmetic. Two approaches to correcting the bias are presented. The Negligible Skew Method described in Appendix B.5.2.2 is appropriate for the NIST curves, since all were selected to be over prime fields near a power of 2 in size. Each NIST prime has at least 32 leading 1's in its binary representation, and at least 16 of the leftmost (high-order) bits are discarded in each block produced. These two facts imply that there is a small fraction ($\leq 1/2^{32}$) of *outlen* outputs for which a bias to 0 may occur in one or more bits. This can only happen when the first 32 bits of an x-coordinate are all zero. As the leftmost 16 bits (at least) are discarded, an adversary can never be certain when a "biased" block has occurred. Thus, any bias due to the modular arithmetic may safely be ignored.~~

E.1.2 Adjusting for the missing bit(s) of entropy in the x coordinates.

In a truly random sequence, it should not be possible to predict any bits from previously observed bits. With the **Dual_EC_DRBG (...),** the full output block of bits produced by the algorithm is "missing" some entropy. Fortunately, by discarding some of the bits, those bits remaining can be made to have nearly "full strength", in the sense that the entropy that they are missing is negligibly small.

To illustrate what can happen, suppose that a mod p curve with m = 256 is selected, and that all 256 bits produced were output by the generator, i.e. that *outlen* = 256 also. Suppose also that 255 of these bits are published, and the 256-th bit is kept "secret". About ½ the time, the unpublished bit could easily be determined from the other 255 bits. Similarly, if

254 of the bits are published, about ¼ of the time the other two bits could be predicted. This is a simple consequence of the fact that only about 1/2 of all 2^m bitstrings of length m occur in the list of all x coordinates of curve points.

The "abouts" in the preceding example can be made more precise, taking into account the difference between 2^m and p, and the actual number of points on the curve (which is always within $2 * p^{½}$ of p). For the NIST curves, these differences won't matter at the scale of the results, so they will be ignored. This allows the heuristics given here to work for any curve with "about" $(2^m)/f$ points, where $f = 1$ is the curve's cofactor.

The basic assumption needed is that the approximately $(2^m)/(2f)$ x coordinates that do occur are "uniformly distributed": a randomly selected m-bit pattern has a probability $1/2f$ of being an x coordinate. The assumption allows a straightforward calculation,--albeit approximate--for the entropy in the rightmost (least significant) m-d bits of **Dual_EC_DRBG** output, with $d << m$.

The formula is $E = -\sum_{j=0}^{2^d} \left[2^{m-d}\, binomprob\left(2^d, z, 2^d - j\right)\right] p_j \log_2 p_j$, where E is the entropy.

The term in braces represents the approximate number of (m-d)-bitstrings that fall into one of $1+2^d$ categories as determined by the number of times j it occurs in an x coordinate; $z = (2f-1)/2f$ is the probability that any particular string occurs in an x coordinate; $p_j = (j*2f)/2^m$ is the probability that a member of the j-th category occurs. Note that the j=0 category contributes nothing to the entropy (randomness).

The values of E for d up to 16 are:

log2(f): 0 d: 0 entropy: 255.00000000 m-d: 256
log2(f): 0 d: 1 entropy: 254.50000000 m-d: 255
log2(f): 0 d: 2 entropy: 253.78063906 m-d: 254
log2(f): 0 d: 3 entropy: 252.90244224 m-d: 253
log2(f): 0 d: 4 entropy: 251.95336161 m-d: 252
log2(f): 0 d: 5 entropy: 250.97708960 m-d: 251
log2(f): 0 d: 6 entropy: 249.98863897 m-d: 250
log2(f): 0 d: 7 entropy: 248.99434222 m-d: 249
log2(f): 0 d: 8 entropy: 247.99717670 m-d: 248
log2(f): 0 d: 9 entropy: 246.99858974 m-d: 247
log2(f): 0 d: 10 entropy: 245.99929521 m-d: 246
log2(f): 0 d: 11 entropy: 244.99964769 m-d: 245
log2(f): 0 d: 12 entropy: 243.99982387 m-d: 244

log2(*f*): 0 *d*: 13 entropy: 242.99991194 *m-d*: 243

log2(f): 0 *d*: 14 entropy: 241.99995597 *m-d*: 242

log2(*f*): 0 *d*: 15 entropy: 240.99997800 *m-d*: 241

log2(*f*): 0 *d*: 16 entropy: 239.99998900 *m-d*: 240

Observations:

a) The table starts where it should, at 1 missing bit;

b) The missing entropy rapidly decreases;

c) For log2(*f*) = 0, i.e, the mod *p* curves, *d*=13 leaves 1 bit of information in every 10,000 (*m*-13)-bit outputs (i.e., one bit of entropy is missing in a collection of 10,000 outputs).

Based on these calculations, for the mod *p* curves, it is recommended that an implementation **shall** remove at least the **leftmost** (most significant) 13 bits of every *m*-bit output.

For ease of implementation, the value of *d* **should** be adjusted upward, if necessary, until the number of bits remaining , *m-d= outlen,* is a multiple of 8. By this rule, the recommended number of bits discarded from each *x*-coordinate will be either 16 or 17. As noted in Section 10.3.1.4, an implementation may decide to truncate additional bits from each *x*-coordinate, provided that the number retained is a multiple of 8.

Because only half of all values in [0, 1, ..., *p*-1] are valid *x*-coordinates on an elliptic curve defined over F_p, it is clear that full *x*-coordinates **should not** be used as pseudorandom bits. The solution to this problem is to truncate these *x*-coordinates by removing the high order 16 or 17 bits. The entropy loss associated with such truncation amounts has been demonstrated to be minimal (see the above chart).

One might wonder if it would be desirable to truncate more than this amount. The obvious drawback to such an approach is that increasing the truncation amount hinders the already sluggish performance. However, there is an additional reason that argues against increasing the truncation. Consider the case where the low *s* bits of each *x*-coordinate are kept. Given some subinterval *I* of length 2^s contained in [0, *p*), and letting N(*I*) denote the number of *x*-coordinates in *I*, recent results on the distribution of *x*-coordinates in [0, *p*) provide the following bound:

$$| N(I) / (p/2) - 2^s / p | < k * \log^2 p / \text{sqrt } p,$$

where *k* is some constant derived from the asymptotic estimates given in [Shparlinski]. For the case of P-521, this is roughly equivalent to:

$$| N(I) - 2^{(s-1)} | < k * 2^{277},$$

where the constant *k* is independent of the value of *s*. For $s < 2^{277}$, this inequality is weak

and provides very little support for the notion that these truncated x-coordinates are uniformly distributed. On the other hand, the larger the value of s, the sharper this inequality becomes, providing stronger evidence that the associated truncated x-coordinates are uniformly distributed. Therefore, by keeping truncation to an acceptable minimum, the performance is increased, and certain guarantees can be made about the uniform distribution of the resulting truncated quantities.

Appendix F: (Informative) Example Pseudocode for Each DRBG

The internal states in these examples are considered to be an array of states, identified by *state_handle*. A particular state is addressed as *internal_state* (*state_handle*), where the value of *state_handle* begins at 0 and ends at *n*-1, and *n* is the number of internal states provided by an implementation. A particular element in the internal state is addressed by *internal_state* (*state_handle*).*element*. In an empty internal state, all bitstrings are set to *Null*, and all integers are set to 0.

For each example in this appendix, arbitary values have been selected that are consistent with the allowed values for each DRBG, as specified in the appropriate table in Section 10.

The pseudocode in this ~~annex~~ appendix does not include the necessary conversions (e.g., integer to bitstring) for an implementation. When conversions are required, they must be accomplished as specified in Appendix B.

The following routine is defined for these pseudocode examples:

Find_state_space (): A function that finds an unused internal state. The function returns a *status* (either "Success" or a message indicating that an unused internal state is not available) and, if *status* = "Success", a *state_handle* that points to an available *internal_state* in the array of internal states. If *status* ≠ "Success", an invalid *state_handle* is returned.

When the uninstantantiate function is invoked in the following examples, the function specified in Section 9.4 is called.

F.1 Hash_DRBG Example

This example of **Hash_DRBG** uses the SHA-1 hash function, and prediction resistance is supported in the example. Both a personalization string and additional input are allowed. A 32-bit incrementing counter is used as the nonce for instantiation (*instantiation_nonce*); the nonce is initialized when the DRBG is installed (e.g., by a call to the clock or by setting it to a fixed value) and is incremented for each instantiation.

A total of 10 internal states are provided (i.e., 10 instantiations may be handled simultaneously).

For this implementation, the functions and algorithms are "inline", i.e., the algorithms are not called as separate routines from the function envelopes. Also, the **Get_entropy_input** function uses only two input parameters, since the first two parameters (as specified in Section 9) have the same value.

The internal state contains values for *V*, *C*, ~~*previous_output_block*~~, *reseed_counter*, *security_strength* and *prediction_resistance_flag*, where *V* and C are bitstrings, and *reseed_counter*, *security_strength* and the *prediction_resistance_flag* are integers. A requested prediction resistance capability is indicated when *prediction_resistance_flag* = 1.

In accordance with Table 2 in Section 10.1, the 112 and 128 bit security strengths may be

supported. Using SHA-1, the following definitions are applicable for the instantiate, generate and reseed functions and algorithms:

1. *highest_supported_security_strength* = 128.
2. Output block length (*outlen*) = 160 bits.
3. Required minimum entropy for instantiation and reseed = *security_strength*.
4. Seed length (*seedlen*) = 440 bits.
5. Maximum number of bits per request (*max_number_of_bits_per_request*) = 5000 bits.
6. Reseed interval (*reseed_interval*) = 100,000 requests.
7. Maximum length of the personalization string (*max_personalization_string_length*) = 512 bits.
8. Maximum length of additional_input (*max_additional_input_string_length*) = 512 bits.
9. Maximum length of entropy input (*max _length*) = 1000 bits.

F.1.1 Instantiation of Hash_DRBG

This implementation will return a text message and an invalid state handle (-1) when an error is encountered. Note that the value of *instantiation_nonce* is an internal value that is always available to the instantiate function.

Note that this implementation does not check the *prediction_resistance_flag*, since the implementation can handle prediction resistance. However, if a consuming application actually wants prediction resistance, the implementation expects that *prediction_resistance_flag* = 1 during instantiation; this will be used in the generate function in Appendix~~x~~ F.1.3.

Instantiate_Hash_DRBG (...):

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

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

Process:

Comment: Check the input parameters.

1. If (*requested_instantiation_security_strength* > 128), then **Return** ("Invalid *requested_instantiation_security_strength*", -1).
2. If (**len** (*personalization_string*) > 512), then **Return** ("*Personalization_string* too long", -1).

Comment: Set the *security_strength* to one of

the valid security strengths.

3. If (*requested_instantiation_security_strength* ≤ 112), then *security_strength* = 112

   Else *security_strength* = 128.

   Comment: Get the *entropy_input*.

4. (*status*, *entropy_input*) = **Get_entropy_input** (*security_strength*, 1000).

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

   Comment: Increment the nonce; actual coding must ensure that it wraps when it's storage limit is reached.

6. *instantiation_nonce* = *instantiation_nonce* + 1.

   Comment: The instantiate algorithm is provided in steps 7-1411.

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

8. *seed* = **Hash_df** (*seed_material*, 440).

9. *V* = *seed*.

10. *C* = **Hash_df** ((0x00 || *V*), 440).

    Comment: Generate the initial block for comparing with the first DRBG output block (for continuous testing).

11. *previous_output_block* = *V*.

12. *H* = **Hash** (0x03 || *V*).

13. *V* = (*V* + *H* + *C* + 1) mod $2^{seedlen}$.

141. *reseed_counter* = 21.

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

1512. (*status*, *state_handle*) = **Find_state_space** ( ).

1613. If (*status* ≠ "Success"), then **Return** (*status*, -1).

1714. *internal_state* (*state_handle*) = {*V*, *C*, *previous_output_block*, *reseed_counter*, *security_strength*, *prediction_resistance_flag*}.

~~18~~15. **Return** ("Success", *state_handle*).

F.1.2 Reseeding a Hash_DRBG Instantiation

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

Reseed_Hash_DRBG_Instantiation (...):

Input: integer *state_handle,* bitstring *additional_input.*

Output: string *status.*

Process:

Comment: Check the validity of the *state_handle*.

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

Comment: Get the internal state values needed to determine the new internal state.

2. Get the appropriate *internal_state* values, e.g., *V* = *internal_state*(*state_handle*).*V*, *security_strength* = *internal_state*(*state_handle*).*security_strength*.

Check the length of the *additional_input*.

3. If (**len** (*additional_input*) > 512), then **Return** ("*Additional_input* too long").

Comment: Get the *entropy_input*.

4. (*status, entropy_input*) = **Get_entropy_input** (*security_strength,* 1000).
5. If (*status* ≠ "Success"), then **Return** ("~~Failure indication returned by~~Catastrophic failure of the *entropy_input* source:" || *status*).

Comment: The reseed algorithm is provided in steps 6-~~10~~10.

6. *seed_material* = 0x01 || *V* || *entropy_input* || *additional_input.*
7. *seed* = **Hash_df** (*seed_material*, 440).
8. *V* = *seed.*
9. *C* = **Hash_df** ((0x00 || *V*), 440).
10. *reseed_counter* = 1.

Comment: Update the *working_state* portion of the internal state.

11. Update the appropriate *state* values.

    11.1 *internal_state* (*state_handle*).*V* = *V*.

    11.2 *internal_* state (*state_handle*).*C* = *C*.

    11.3 *internal_ state* (*state_handle*).*reseed_counter* = *reseed_counter*.

12. **Return** ("Success").

F.1.3 Generating Pseudorandom Bits Using Hash_DRBG

The implementation returns a *Null* string as the pseudorandom bits if an error has been detected. Prediction resistance is requested when *prediction_resistance_request* = 1.

In this implementation, prediction resistance is requested by supplying *prediction_resistance_request* = 1 when the **Hash_DRBG** function is invoked.

Hash_DRBG (...):

Input: integer (*state_handle*, *requested_no_of bits*, *requested_security_strength*, *prediction_resistance_request*), bitstring *additional_input.*

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

Process:

Comment: Check the validity of the *state_handle*.

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

Comment: Get the internal state values.

2. *V* = *internal_state* (*state_handle*).*V*, *C* = *internal_state* (*state_handle*).*C*, ~~*previous_output_block = internal_state (state_handle).previous_output_block.*~~ *reseed_counter* = *internal_state* (*state_handle*).*reseed_counter*, *security_strength* = *internal_state* (*state_handle*).*security_strength*, *prediction_resistance_flag* = *internal_state* (*state_handle*).*prediction_resistance_flag*.

Comment: Check the validity of the other input parameters.

3. If (*requested_no_of_bits* > 5000) then **Return** ("Too many bits requested", *Null*).

4. If (*requested_security_strength* > *security_strength*), then **Return** ("Invalid *requested_security_strength*", *Null*).

5. If (**len** (*additional_input*) > 512), then **Return** ("*Additional_input* too long",

Null).

6. If ((*prediction_resistance_request* = 1) and (*prediction_resistance_flag* ≠ 1)), then **Return** ("Prediction resistance capability not instantiated", *Null*).

   Comment: Reseed if necessary. Note that since the instantiate algorithm is inline with the functions, this step has been written as a combination of steps 6 and 7 of Section 9.3 and step 1 of the generate algorithm in Section 10.1.1.4. Because of this combined step, step 9 of Section 9.3.is not required.

7. If ((*reseed_counter* > 100,000) OR (*prediction_resistance_request* = 1)), then

   7.1 *status* = **Reseed_ Hash_DRBG_Instantiation** (*state_handle, additional_input*).

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

   Comment: Get the new internal state values that have changed.

   7.3 *V* = *internal_state* (*state_handle*).*V*, *C* = *internal_state* (*state_handle*).*C*, *reseed_counter* = *internal_state* (*state_handle*).*reseed_counter*.

   7.4 *additional_input* = *Null*.

   Comment: Steps 8-16 provide the rest of the generate algorithm. Note that in this implementation, the **Hashgen** routine is also inline as steps 9-13.

8. If (*additional_input* ≠ *Null*), then do

   7.1 *w* = **Hash** (0x02 || *V* || *additional_input*).

   7.2 $V = (V + w) \bmod 2^{440}$.

9. $m = \left\lceil \frac{requested\_no\_of\_bits}{outlen} \right\rceil$.

10. *data* = *V*.

11. *W* = the Null string.

12. For *i* = 1 to *m*

    12.1 w_i = **Hash** (*data*).

    12.22 ~~If (w_i = *previous_output_block*), then **Return** ("Fatal error: output blocks match", *Null*).~~

~~12.3 *previous_output_block* = w_i.~~

~~12.4~~ $W = W \| w_i$.

12.~~5~~3 $data = (data + 1) \bmod 2^{440}$.

13. *pseudorandom_bits* = Leftmost (*requested_no_of_bits*) bits of *W*.

14. H = **Hash** (0x03 || V).

15. $V = (V + H + C + reseed\_counter) \bmod 2^{440}$.

16. *reseed_counter* = *reseed_counter* + 1.

Comments: Update the *working_state*.

13. Update the changed values in the *state*.

13.1 *internal_state* (*state_handle*).$V = V$.

13.2 ~~*internal_state (state_handle).previous_output_block = previous_output_block.*~~

~~13.3~~ *internal_state* (*state_handle*).*reseed_counter* = *reseed_counter*.

14. **Return** ("Success", *pseudorandom_bits*).

F.2 HMAC_DRBG Example

This example of **HMAC_DRBG** uses the SHA-256 hash function. Reseeding and prediction resistance are not provided. The nonce for instantiation consists of a random value with *security_strength*/2 bits of entropy; the nonce is obtained by increasing the call for entropy bits via the **Get_entropy_input** call by *security_strength*/2 bits (i.e., by adding *security_strength*/2 bits to the *security_strength* value). The **Update** function is specified in Section 10.1.2.2.

A personalization string is allowed, but additional input is not. A total of 3 internal states are provided. For this implementation, the functions and algorithms are written as separate routines. Also, the **Get_entropy_input** function uses only two input parameters, since the first two parameters (as specified in Section 9) have the same value.

The internal state contains the values for *V*, *Key*, *reseed_counter*, and *security_strength*, where *V* and *C* are bitstrings, and *reseed_counter* and *security_strength* are integers.

In accordance with Table 2 in Section 10.1, security strengths of 112, 128, 192 and 256 may be supported. Using SHA-256, the following definitions are applicable for the instantiate and generate functions and algorithms:

1. *highest_supported_security_strength* = 256.
2. Output block (*outlen*) = 256 bits.
3. Required minimum entropy for the entropy input at instantiation = 3/2 *security_strength* (this includes the entropy required for the nonce).

4. Seed length (*seedlen*) = 440 bits.
5. Maximum number of bits per request (*max_number_of_bits_per_request*) = 7500 bits.
6. Reseed_interval (*reseed_ interval*) = 10,000 requests.
7. Maximum length of the personalization string (*max_personalization_string_length*) = 160 bits.
8. Maximum length of the entropy input (*max _length*) = 1000 bits.

F.2.1 Instantiation of HMAC_DRBG

This implementation will return a text message and an invalid state handle (-1) when an error is encountered.

Instantiate_HMAC_DRBG (...):

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

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

Process:

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*) > 160), then **Return** ("*Personalization_string* too long", -1)

Comment: Set the *security_strength* to one of the valid security strengths.

3. If (*requested_security_strength* ≤ 112), then *security_strength* = 112

   Else (*requested_ security_strength* ≤ 128), then *security_strength* = 128

   Else (*requested_ security_strength* ≤ 192), then *security_strength* = 192

   Else *security_strength* = 256.

Comment: Get the *entropy_input and the nonce*.

4. *min_entropy* = 1.5 × *security_strength*.
5. (*status*, *entropy_input*) = **Get_entropy_input** (*min_entropy*, 1000).
6. If (*status* ≠ "Success"), then **Return** ("~~Failure indication returned~~Catastrophic failure of ~~by~~ the entropy source:" || *status*, -1).

Comment: Invoke the instantiate algorithm.

Note that the *entropy_input* contains the nonce.

7. (*V*, *Key*, *reseed_counter*) = **Instantiate_algorithm** (*entropy_input*, *personalization_string*).

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

8. (*status*, *state_handle*) = **Find_state_space** ( ).
9. If (*status* ≠ "Success"), then **Return** ("No available state space:" || *status*, -1).
10. *internal_state* (*state_handle*) = {*V*, *Key*, *reseed_counter*, *security_strength*}.
11. Return ("Success" and *state_handle*).

Instantiate_algorithm (...):

Input: bitstring (*entropy_input*, *personalization_string*).

Output: bitstring (*V*, *Key*), integer *reseed_counter*.

Process:

1. *seed_material* = *entropy_input* || *personalization_string*.
2. Set *Key* to *outlen* bits of zeros.
3. Set *V* to *outlen*/8 bytes of 0x01.
4. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).
5. ~~(*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).~~
7. ~~*reseed_counter* = 1.~~
8. **Return** (*V*, *Key*, *reseed_counter*).

F.2.2 Generating Pseudorandom Bits Using HMAC_DRBG

The implementation returns a *Null* string as the pseudorandom bits if an error has been detected.

HMAC_DRBG(...):

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

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

Process:

Comment: Check for a valid state handle.

1. If ((*state_handle* < 0) or (*state_handle* > 2) or (*internal_state* (*state_handle*) = {*Null*, *Null*, 0, 0}), then **Return** ("State not available for the indicated *state_handle*", *Null*).

Comment: Get the internal state.

2. *V* = *internal_state* (*state_handle*).*V*, *Key* = *internal_state* (*state_handle*).*Key*, *security_strength* = *internal_state* (*state_handle*).*security_strength*, *reseed_counter* = *internal_state (state_handle).reseed_counter.*

   Comment: Check the validity of the rest of the input parameters.

3. If (*requested_no_of_bits* > 7500), then **Return** ("Too many bits requested", *Null*).

4. If (*requested_security_strength* > *security_strength*), then **Return** ("Invalid *requested_security_strength*", *Null*).

   Comment: Invoke the generate algorithm.

5. (*status*, *pseudorandom_bits*, *V*, *Key*, *reseed_counter*) = **Generate_algorithm** (*V*, *Key*, *reseed_counter*, *requested_number_of_bits*).

6. If (*status* = "Reseed required"), then **Return** ("DRBG can no longer be used. Please re-instantiate or reseed", *Null*).

7. ~~If (*status* = "ERROR: outputs match"), then~~

   ~~8.1 For *i* = 0 to 3, do~~

   ~~8.1.1 *status* = **Uninstantiate** (*i*).~~

   ~~8.1.2 If (*status* ≠ "Success"), then **Return** ("DRBG FAILURE: Successive outputs match, and uninstantiate failed", *Null*).~~

   ~~8.2 **Return** ("DRBG " || *status*, *Null*).~~

   ~~Comment: Update the internal state.~~

~~9.~~ *internal_state* (*state_handle*) = {*V*, *Key*, *security_strength*, *reseed_counter*}.

~~10~~8. **Return** ("Success", *pseudorandom_bits*).

Generate_algorithm (...):

Input: bitstring (*V*~~_old~~, *Key*), integer (*reseed_counter*, *requested_number_of_bits*).

Output: string *status*, bitstring (*pseudorandom_bits*, *V*, *Key*), integer *reseed_counter*.

Process:

1 If (*reseed_counter* ≥ 10,000), then **Return** ("Reseed required", *Null, V, Key, reseed_counter*).

2. *temp* = *Null*.

3 While (**len** (*temp*) < *requested_no_of_bits*) do:

3.1 *V* = **HMAC** (*Key*, *V*).

3.2 ~~If (*V = V_old*), then **Return** ("ERROR: outputs match", *Null*, *V*, *Key*, *reseed_counter*).~~

~~3.3 *V_old = V*.~~

~~3.4 *temp = temp* || *V*.~~

4. *pseudorandom_bits* = Leftmost (*requested_no_of_bits*) of *temp*.
5. (*Key*, *V*) = **Update** (*additional_input*, *Key*, *V*).
6. *reseed_counter* = *reseed_counter* + 1.
7. **Return** ("Success", *pseudorandom_bits*, *V*, *Key*, *reseed_counter*).

F.3 CTR_DRBG Example Using a Derivation Function

This example of **CTR_DRBG** uses AES-128. The reseed and prediction resistance capabilities are available, and a block cipher derivation function using AES-128 is used. Both a personalization string and additional input are allowed. A total of 5 internal states are available. For this implementation, the functions and algorithms are written as separate routines. The **Block_Encrypt** function (specified in Section 10.4.2) uses AES-128 in the ECB mode.

The nonce for instantiation (*instantiation_nonce*) consists of a 32-bit incrementing counter. The nonce is initialized when the DRBG is installed (e.g., by a call to the clock or by setting it to a fixed value) and is incremented for each instantiation.

The internal state contains the values for *V*, *Key*, ~~*previous_output_block*~~, *reseed_counter*, and *security_strength*, where *V*, and *Key* ~~and *previous_output_block*~~ are strings, and all other values are integers. Since prediction resistance is always available, there is no need for *prediction_resistance_flag* in the internal state.

In accordance with Table 3 in Section 10.2.1, security strengths of 112 and 128 may be supported. Using AES-128, the following definitions are applicable for the instantiate, reseed and generate functions:

1. *highest_supported_security_strength* = 128.
2. Output block length (*outlen*) = 128 bits.
3. Key length (*keylen*) = 128 bits.
4. Required minimum entropy for the entropy input during instantiation and reseeding = *security_strength*.
5. Minimum entropy input length (*min _length*) = *security_strength* bits.
6. Maximum entropy input length (*max _length*) = 1000 bits.
7. Maximum personalization string input length (*max_personalization_string_input_length*) = 800 bits.

8. Maximum additional input length (*max_additional_input_length*) = 800 bits.
9. Seed length (*seedlen*) = 256 bits.
10. Maximum number of bits per request (*max_number_of_bits_per_request*) = 4000 bits.
11. Reseed interval (*reseed_interval*) = 100,000 requests. Note that for this value, the instantiation count will not repeat during the reseed interval.

F.3.1 The Update Function

Update (...):

Input: bitstring (*provided_data*, *Key*, *V*).

Output: bitstring (*Key*, *V*).

Process:

1. *temp* = *Null.*
2. While (**len** (*temp*) < 256) do
   3.1 $V = (V + 1) \bmod 2^{128}$.
   3.2 *output_block* = **AES_ECB_Encrypt** (*Key*, *V*).
   3.3 *temp* = *temp* || *ouput_block.*
4. *temp* = Leftmost 256 bits of *temp.*
5 *temp* = *temp* ⊕ *provided_data.*
6. *Key* = Leftmost 128 bits of *temp.*
7. *V* = Rightmost 128 bits of *temp.*
8. **Return** (*Key*, *V*).

F.3.2 Instantiation of CTR_DRBG Using a Derivation Function

This implementation will return a text message and an invalid state handle (-1) when an error is encountered. **Block_Cipher_df** is the derivation function in Section 10.4.2, and uses AES-128 in the ECB mode as the **Block_Encrypt** function.

Note that this implementation does not include the *prediction_resistance_flag* in the input parameters, nor save it in the internal state, since prediction resistance is always available.

Instantiate_CTR_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* > 128) then **Return** ("Invalid *requested_instantiation_security_strength*", -1).
2. If (**len** (*personalization_string*) > 800), then **Return** ("*Personalization_string* too long", -1).
3. If (*requested_instantiation_security_strength* ≤ 112), then *security_strength* = 112

   Else *security_strength* = 128.

   Comment: Get the entropy input.

4. (*status*, *entropy_input*) = **Get_entropy_input** (*security_strength*, *security_strength*, 1000).
5. If (*status* ≠ "Success"), then **Return** ("~~Failure indication returned by~~Catastrophic failure of the entropy source" || *status*, -1).

   Comment: Increment the nonce; actual coding must ensure that the nonce wraps when its storage limit is reached, and that the counter pertains to all instantiations, not just this one.

6. *instantiation_nonce* = *instantiation_nonce* + 1.

   Comment: Invoke the instantiate algorithm.

7. (*V*, *Key*, ~~*previous_output_block,*~~ *reseed_counter*) = **Instantiate_algorithm** (*entropy_input*, *instantiation_nonce*, *personalization_string*).

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

8. (*status*, *state_handle*) = **Find_state_space** ( ).
9. If (*status* ≠ "Success"), then **Return** ("No available state space:" || *status*, -1).

   Comment: Save the internal state.

10. *internal_state_* (*state_handle*) = {*V*, *Key*, ~~*previous_output_block,*~~ *reseed_counter*, *security_strength*}.
11. **Return** ("Success", *state_handle*).

Instantiate_algorithm (...):

Input: bitstring (*entropy_input*, *nonce*, *personalization_string*).

Output: bitstring (*V*, *Key*), integer (*reseed_counter*).

Process:

1. *seed_material* = *entropy_input* || *nonce* || *personalization_string*.
2. *seed_material* = **Block_Cipher_df** (*seed_material*, 256).
3. *Key* = 0^{128}. Comment: 128 bits.
4. *V* = 0^{128}. Comment: 128 bits.
5. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).
6. *reseed_counter* = 1.
7. ~~*previous_output_block* = **AES_ECB_Encrypt** (*Key*, *V*).~~
8. ~~*zeros* = $0^{seedlen}$. Comment: Produce a string of *seedlen* zeros.~~
9. ~~(*Key*, *V*) = **Update** (*zeros*, *Key*, *V*).~~
10. ~~**Return** (*V*, *Key*, *previous_output_block*, *reseed_counter*).~~

F.3.3 Reseeding a CTR_DRBG Instantiation Using a Derivation Function

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

Reseed_CTR_DRBG_Instantiation (...):

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

Output: string *status*.

Process:

Comment: Check for the validity of *state_handle*.

1. If ((*state_handle* < 0) or (*state_handle* > 4) or (*internal_state*(*state_handle*) = {*Null*, *Null*, ~~*Null*~~, 0, 0}), then **Return** ("State not available for the indicated *state_handle*").

Comment: Get the internal state values.

2. *V* = *internal_state* (*state_handle*).*V*, *Key* = *internal_state* (*state_handle*).*Key*, ~~*previous_output_block* = *internal_state* (*state_handle*).*previous_output_block*,~~ *security_strength* = *internal_state* (*state_handle*).*security_strength*.
3. If (**len** (*additional_input*) > 800), then **Return** ("*Additional_input* too long").
4. (*status*, *entropy_input*) = **Get_entropy_input** (*security_strength*, *security_strength*, 1000).
6. If (*status* ≠ "Success"), then **Return** ("~~Failure indication returned~~

byCatastrophic failure of the entropy source:" || *status*).

Comment: Invoke the reseed algorithm.

7. (*V*, *Key*, *reseed_counter*) = **Reseed_algorithm** (*V*, *Key*, *reseed_counter*, *entropy_input*, *additional_input*).
8. *internal_state* (*state_handle*) = {*V*, *Key*, ~~*previous_output_block,*~~ *reseed_counter*, *security_strength* }.
9. **Return** ("Success").

Reseed_algorithm (...):

Input: bitstring (*V*, *Key*), integer (*reseed_counter*), bitstring (*entropy_input*, *additional_input*).

Output: bitstring (*V*, *Key*), integer (*reseed_counter*).

Process:

1. *seed_material* = *entropy_input* || *additional_input.*
2. *seed_material* = **Block_Cipher_df** (*seed_material*, 256).
3. (*Key*, *V*) = **Update** (*seed_material*, *Key*, *V*).
4. *reseed_counter* = 1.
5. **Return** *V*, *Key*, *reseed_counter*).

F.3.4 Generating Pseudorandom Bits Using CTR_DRBG

The implementation returns a *Null* string as the pseudorandom bits if an error has been detected.

CTR_DRBG(...):

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

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

Process:

Comment: Check the validity of *state_handle*.

1. If ((*state_handle* < 0) or (*state_handle* > 4) or (*internal_state* (*state_handle*) = {*Null*, *Null*, ~~*Null,*~~ 0, 0}), then **Return** ("State not available for the indicated *state_handle*", *Null*).

Comment: Get the internal state.

2. *V* = *internal_state* (*state_handle*).*V*, *Key* = *internal_state* (*state_handle*).*Key*, ~~*previous_output_block = internal_state (state_handle).previous_output_block,*~~ *security_strength* = *internal_state* (*state_handle*).*security_strength*,

reseed_counter = *internal_state (state_handle).reseed_counter*.

Comment: Check the rest of the input parameters.

3. If (*requested_no_of_bits* > 4000), then **Return** ("Too many bits requested", *Null*).
4. If (*requested_security_strength* > *security_strength*), then **Return** ("Invalid *requested_security_strength*", *Null*).
5. If (**len** (*additional_input*) > 800), then **Return** ("*Additional_input* too long", *Null*).
6. *reseed_required_flag* = 0.
7. If ((*reseed_required_flag* = 1) OR (*prediction_resistance_flag* = 1)), then
   7.1 *status* = **Reseed_CTR_DRBG_Instantiation** (*state_handle*, *additional_input*).
   7.2 If (*status* ≠ "Success"), then **Return** (*status*, *Null*).

   Comment: Get the new working state values; the administrative information was not affected.

   7.3 *V* = *internal_state* (*state_handle*).*V*, *Key* = *internal_state* (*state_handle*).*Key*, ~~*previous_output_block* = *internal_state* (*state_handle*).*previous_output_block*,~~ *reseed_counter* = *internal_state (state_handle).reseed_counter*.
   7.4 *additional_input* = *Null*.
   7.5 *reseed_required_flag* = 0.

   Comment: Generate bits using the generate algorithm.

8. (*status*, *pseudorandom_bits*, *V*, *Key*, ~~*previous_output_block*,~~ *reseed_counter*) = **Generate_algorithm** (*V*, *Key*, ~~*previous_output_block*,~~ *reseed_counter*, *requested_number_of_bits*, *additional_input*).
9. If (*status* = "Reseed required"), then
   9.1 *reseed_required_flag* = 1.
   9.2 Go to step 7.
10. ~~If (*status* = "ERROR: outputs match"), then~~
    ~~10.1 For *i* = 0 to 5, do~~
       ~~8.1.1 *status* = **Uninstantiate** (*i*).~~

~~8.1.2 If (*status* ≠ "Success"), then **Return** ("DRBG FAILURE: Successive outputs match, and uninstantiate failed", *Null*).~~

~~10.2 **Return** ("DRBG: " || *status*, *Null*).~~

~~11.~~ *internal_state* (*state_handle*) = {*V*, *Key*, reseed_counter, ~~*previous_output_block, security_strength, reseed_counter*~~).

~~12~~11. **Return** ("Success", *pseudorandom_bits*).

Generate_algorithm (...):

Input: bitstring (*V* ~~*old*~~, *Key* ~~*old, previous_output_block*~~), integer (*reseed_counter*, *requested_number_of_bits*) bitstring *additional_input*.

Output: string *status*, bitstring (*returned_bits*, *V*, *Key*, ~~*previous_output_block*~~), integer *reseed_counter*.

Process:

1. If (*reseed_counter* > 100,000), then **Return** ("Reseed required", *Null*, *V*, *Key*, ~~*previous_output_block, reseed_counter*~~).
2. If (*additional_input* ≠ *Null*), then
   2.1 *additional_input* = **Block_Cipher_df** (*additional_input*, 256).
   ~~2.2 If (*temp* < 256), then *additional_input* = *additional_input* || $0^{256 - temp}$.~~
   ~~2.4 (*Key*, *V*) = **Update** (*additional_input*, *Key_old*, *V*) *old*).~~
   ~~2.5 If((*Key* = *Key_old*) or (*V* = *V_old*)), then **Return** ("ERROR: outputs match", *Null*, *V*, *Key*, *previous_output_block*, *reseed_counter*).~~
3. *temp* = *Null*.
4. While (**len** (*temp*) < *requested_number_of_bits*) do:
   4.1 *V* = (*V* + 1) mod 2^{128}.
   4.2 *output_block* = **AES_ECB_Encrypt** (*Key*, *V*).
   4.3 ~~If (*output_block* = *previous_output_block*), then **Return** ("ERROR: outputs match", *Null*, *V*, *Key*, *previous_output_block*, *reseed_counter*).~~
   ~~4.4 *previous_output_block* = *output_block*.~~
   ~~4.5 *temp* = *temp* || *ouput_block*.~~
5. *returned_bits* = Leftmost (*requested_number_of_bits*) of *temp*.
6. *zeros* = 0^{256}. Comment: Produce a string of 256 zeros.
7. (*Key*, *V*) = **Update** (*zeros*, *Key*, *V*)

8. *reseed_counter* = *reseed_counter* + 1.
9. **Return** ("Success", *returned_bits*, *V*, *Key*, ~~*previous_output_block,*~~ *reseed_counter*).

F.4 CTR_DRBG Example Without a Derivation Function

This example of **CTR_DRBG** is the same as the previous example except that a derivation function is not used (i.e., full entropy is always available). As in Appendix F.3, the CTR_DRBG uses AES-128. The reseed and prediction resistance capabilities are available. Both a personalization string and additional input are allowed. A total of 5 internal states are available. For this implementation, the functions and algorithms are written as separate routines. The **Block_Encrypt** function (as specified in Section 10.4.2) uses AES-128 in the ECB mode.

The nonce for instantiation (*instantiation_nonce*) consists of a 32-bit incrementing counter that is the initial bits of the personalization string (Section 8.6.1 states that when a derivation function is used, the nonce, if used, is contained in the personalization string). The nonce is initialized when the DRBG is installed (e.g., by a call to the clock or by setting it to a fixed value) and is incremented for each instantiation.

The internal state contains the values for *V*, *Key*, ~~*previous_output_block,*~~ *reseed_counter*, and *security_strength*, where *V*, ~~,~~ and *Key* ~~and *previous_output_block*~~ are strings, and all other values are integers.Since prediction resistance is always available, there is no need for *prediction_resistance_flag* in the internal state.

In accordance with Table 3 in Section 10.2.1, security strengths of 112 and 128 may be supported. The definitions are the same as those provided in Appendix F.3, except that to be compliant with Table 3, the maximum size of the *personalization_string* is 224 bits in order to accommodate the 32-bits of the *instantiation_nonce* (i.e., **len** (*instantiation_nonce*) + **len** (*personalization_string*) must be ≤ *seedlen*, where *seedlen* = 256 bits). In addition, the maximum size of any *additional_input* is 256 bits (i.e., len (*additional_input* ≤ *seedlen*)).

F.4.1 The Update Function

The update function is the same as that provided in Appendix F.3.1.

F.4.2 Instantiation of CTR_DRBG Without a Derivation Function

The instantiate function (**Instantiate_CTR_DRBG**) is the same as that provided in Appendix F.3.2, except for the following:

- Step 2 is replaced by:

  If (**len** (*personalization_string*) > 224), then **Return** ("*Personalization_string* too long", -1).

- Step 6 is replaced by :

instantiation_nonce = *instantiation_nonce* + 1.

personalization_string = *instantiation_nonce* || *personalization_string*.

The instantiate algorithm (**Instantiate_algorithm**) is the same as that provided in Appendix F.3.2, except that step 1 is replaced by:

temp = **len** (*personalization_string*).

If (*temp* < 256), then *personalization_string* = *personalization_string* || $0^{256-temp}$.

seed_material = *entropy_input* ⊕ *personalization_string*.

F.4.3 Reseeding a CTR_DRBG Instantiation Without a Derivation Function

- The reseed function (**Reseed_CTR_DRBG**) is the same as that provided in Appendix F.3.3, except that step 3 is replaced by:

  If (**len** (*additional_input*) > 256), then **Return** ("*Additional_input* too long").

The instantiate algorithm (**Reseed_algorithm**) is the same as that provided in Appendix F.3.3, except that step 1 is replaced by:

temp = **len** (*additional_input*).

If (*temp* < 256), then *additional_input* = *additional_input* || $0^{256-temp}$.

seed_material = *entropy_input* ⊕ *additional_input*.

F.4.4 Generating Pseudorandom Bits Using CTR_DRBG

The generate function (**CTR_DRBG**) is the same as that provided in Appendix F.3.4, except that step 5 is replaced by :

If (**len** (*additional_input*) > 256), then **Return** ("*Additional_input* too long", *Null*).

The generate algorithm (**Generate_algorithm**) is the same as that provided in Appendix F.3.4, except that step 2.1 is replaced by:

temp = **len** (*additional_input*).

If (*temp* < 256), then *additional_input* = *additional_input* || $0^{256-temp}$.

~~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-521

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. Also, the **Get_entropy_input** function uses only two input parameters, since the first two parameters (the *min_entropy* and the *min_length*) have the same value.

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. Maximum entropy input length (*max _length*) = 1000 bits.
5. Maximum personalization string length (*max_personalization_string_length*) = 800 bits.
6. Maximum additional input length (*max_additional_input_length*) = 800 bits.
7. Seed length (*seedlen*): = 2 × *security_strength*.
8. Maximum number of bits per request (*max_number_of_bits_per_request*) = 1000 bits.
9. 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* = 224; *outlen* = 240}

   Else if (*requested_instantiation_security_strength* ≤ 128), then

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

   Else if (*requested_instantiation_security_strength* ≤ 192), then

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

   Else {*security_strength* = 256;, *seedlen* = 512; *outlen* = 504}.
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*, 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, 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*, 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 ($block\_counter + \left\lceil \frac{requested\_number\_of\_bits}{outlen} \right\rceil > 10.0002^{32}$), 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* || (**rightmost** *outlen* 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. *internal_state.s = s.*

22. *internal_state.r_old = r_old.*

23. *internal_state.block_counter = block_counter.*

24. **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 an available hash function, 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 security of the DRBG is dependent 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 consuming application, but there may be restrictions that need to be addressed (e.g., code size or performance requirements).

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

G.1 Hash_DRBG

Hash_DRBG is based on the use of an Approved hash function in a counter mode similar to the counter mode specified in NIST SP 800-38A. For each Generate request, the current value of V (a secret value in the internal state) is used as the starting counter that is iteratively changed to generate each successive n-bit block of requested output, where n is the number of bits in the hash function output block. At the end of the Generate request, and before the pseudorandom output is returned to the consuming application, the secret

value *V* is updated in order to prevent backtracking.

Performance. Within a Generate request, each *n*-bit block of output requires one hash function computation and some additions; an additional hash function computation is required to provide the backtracking resistance. **Hash_DRBG** produces pseudorandom output bits in about half the time required by **HMAC_DRBG**.

Security. **Hash_DRBG**'s security depends on the underlying hash function's behavior when processing a series of sequential input blocks. If the hash function is replaced by a random oracle, **Hash_DRBG** is secure. It is difficult to relate the properties of the hash function required by **Hash_DRBG** with common properties, such as collision resistance, pre-image resistance, or pseudorandomness. There are known problems with **Hash_DRBG** when the DRBG is instantiated with insufficient entropy for the requested security strength, and then later provided with enough entropy to attain the amount of entropy required for the security strength, via the inclusion of additional input during a Generate request. However, these problems do not affect the DRBG's security when **Hash_DRBG** is instantiated with the amount of entropy specified in this Recommendation.

Constraints on Outputs. As shown in Table 2 of Section 10.1, for each hash function, up to 2^{48} generate requests may be made, each of up to 2^{19} bits.

Resources. **Hash_DRBG** requires access to a hash function, and the ability to perform addition with *seedlen*-bit integers. **Hash_DRBG** uses the hash-based derivation function, **Hash_df** specified in Section 10.4.1 during instantiation and reseeding. Any implementation requires the storage space required for the internal state (see Section 10.1.1.1).

Algorithm Choices. The choice of hash functions that may be used by **Hash_DRBG** is discussed in Section 10.1.

~~[Need to insert text here]~~

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 = **HMAC** (*Key*, *V*).

At the end of a generation request, the DRBG generates a new *Key* and *V*, each requiring one HMAC computation.

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 consuming applications are expected to need output bits faster than **HMAC_DRBG** can provide them.

Security. The security of **HMAC_DRBG** is based on the assumption that an approved 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** ultimately relies on the pseudorandomness properties of the underlying hash function. Note that, 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, and the **HMAC_DRBG** requires far less demanding properties of the underlying hash function than **Hash_DRBG**.

Constraints on Outputs. As shown in Table 2 of Section 10.1, for each hash function, up to 2^{48} generate requests may be made, each of up to 2^{19} bits.

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. ***HMAC_DRBG*** *requires access to* a dedicated HMAC implementation for optimal performance. However, a general-purpose hash function implementation can always be used to implement HMAC. Any implementation requires the storage space required for the internal state (see Section 10.1.2.1)a hashing engine or an HMAC implementation, and the storage space required for the internal state (see Section 10.1.2.1.

Algorithm Choices. The choice of algorithms hash functions that may be used by **HMAC_DRBG** is discussed in Section 10.1.

G.3 CTR_DRBG

CTR_DRBG is a DRBG based on using an Approved block cipher algorithm in counter mode (see SP 800-38A). At the present time of this writing, only three-key TDEA and AES are approved for use within ANS X9.82by the Federal government for use in this DRBG. 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.

Performance. For large Generate requests, **CTR_DRBG** produces outputs at the same speed as the underlying block cipher algorithm encrypts data. Furthermore, **CTR_DRBG** is parallelizeable. At the end of each Generate request, work equivalent to 2, 3 or 4 encryptions is performed, depending on the choice of underlying block cipher algorithm, to generate new keys and counters for the next Generate request.

Security. The security of **CTR_DRBG** is directly based on the security of the underlying block cipher algorithm, 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 algorithm.

Constraints on Outputs. AsFor shown in Table 3 of Section 10.2.1, 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. However, the smaller block size of TDEA imposes more 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 2, 3 and 4 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.

When a derivation function is used, **CTR_DRBG** can process the personalization string and any additional input in the same way as any other DRBG, but at a cost in performance because of the use of the derivation function. Such an implementation may be seeded by any Approved source of entropy input that may or may not provide full entropy.

When a derivation function is not used, **CTR_DRBG** is more efficient when the personalization string and any additional input are provided, but is less flexible because the lengths of the personalization string and additional input cannot exceed *seedlen* bits. Such implementations must be seeded by a source of entropy input that provides full entropy (e.g., an Approved conditioned entropy source or Approved RBG).

CTR_DRBG requires access to a block cipher algorithm, including the ability to change keys, and the storage space required for the internal state (see Section 10.2.1.1).

~~**CTR_DRBG** is ideal for situations in which 1) prediction resistance is often required, and 2) an Approved entropy source or another RBG is readily available to provide entropy input so that a derivation function is not required. Without the readily available source of entropy input, a derivation function must be used each time additional entropy input is required, thus slowing down the random bit generation process. For instantiation and reseeding without frequent requests for prediction resistance, however, the use of a derivation function should not lead to an important performance penalty, since both these operations are done only very rarely. **CTR_DRBG** implementations may also suffer a~~

~~substantial performance penalty if they process additional input with generate requests, since the derivation function may be required in this case as well, unless the length of the additional input is limited to be less than or equal to the seed length (*seedlen*).~~
~~**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 10.2.1.1).~~

Algorithm Choices.~~:~~ ~~-~~The choice of block cipher algorithms and key sizes that may be used by **CTR_DRBG** is discussed in Section 10.2.1.

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 $R*Q$.

Formatted: Font color: Auto

Performance. Due to the elliptic curve arithmetic involved in this DRBG, this algorithm generates pseudorandom bits more slowly than the other DRBGS in this Recommendation. 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 a substantial increase in 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 where 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.

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 Recommendation. The **Dual_EC_DRBG** is the only

DRBG in this Recommendation whose security is related to a hard problem in number theory.

Constraints on Outputs. 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, where the size of the output blocks is discussed in Section 10.3.1.4. Since the sequence of output blocks is 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 three possible curves.

Resources. Any entropy input source may be used with **Dual_EC_DRBG**, provided that it is capable of generating at least *min_entropy* bits of entropy in a string of max_length = 2^{13} 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.

~~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 (2m).~~

~~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.~~

~~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 10,000232 times the block length. The smallest blocksize is 216, meaning that at least 2M bits can be requested on each call.).~~

Formatted: Font: French (France), Highlight
Formatted: Font: Not Bold, French (France), Highlight
Formatted: Font: French (France), Highlight
Formatted: Font: Not Bold, French (France), Highlight
Formatted: Font: French (France), Highlight
Formatted: Font: Not Italic, French (France), Not Superscript/ Subscript, Highlight
Formatted: Font: French (France), Highlight
Formatted: Font: French (France)
Formatted: Font: Not Bold, No underline, French (France), Highlight
Formatted: Font: French (France), Highlight
Formatted: Font: Not Italic, French (France), Highlight
Formatted: Font: French (France), Highlight
Formatted: Font color: Auto, French (France), Highlight
Formatted: Font color: Auto, French (France), Not Superscript/ Subscript, Highlight
Formatted: French (France), Highlight
Formatted: Font color: Auto, French (France), Highlight
Formatted: French (France), Highlight
Formatted: French (France)

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

~~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.~~

Formatted: French (France), Highlight

Formatted: Font: Not Bold, French (France), Highlight

Formatted: French (France), Highlight

Formatted: Font: Not Bold, French (France), Highlight

Formatted: French (France), Highlight

Formatted: Font: Not Italic, French (France), Highlight

Formatted: French (France), Highlight

Formatted: Font: Not Bold, French (France), Highlight

Formatted: French (France), Highlight

Formatted: Font: Not Italic, French (France), Highlight

Formatted: French (France), Highlight

Formatted: French (France)

Appendix H : (Informative) References

Federal Information Processing Standard 140-2, *Security Requirements for Cryptographic Modules*, May 25, 2001.

Federal Information Processing Standard 180-2, *Secure Hash Standard* (*SHS*), August 2002.

Federal Information Processing Standard 186-3, *Digital Signature Standard* (*DSS*), [Date to be inserted].

Federal Information Processing Standard 197, *Advanced Encryption Standard* (*AES*), November 2001.

Federal Information Processing Standard 198, *Keyed-Hash Message Authentication Code* (*HMAC*), March 6, 2002.

National Institute of Standards and Technology Special Publication (SP) 800-38A, *Recommendation for Block Cipher Modes of Operation - Methods and Techniques*, December 2001.

NIST Special Publication (SP) 800-57, Part 1, *Recommendation for Key Management*: General, [August 2005].

NIST Special Publication (SP) 800-67, *Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher*, May 2004.

American National Standard (ANS) X9.62-2000, *Public Key Cryptography for the Financial Services Industry - The Elliptic Curve Digital Signature Algorithm* (*ECDSA*).

American National Standard (ANS) X9.63-2000, *Public Key Cryptography for the Financial Services Industry - Key Agreement and Key Transport Using Elliptic Key Cryptography.*

[Shparlinski] Mahassni, Edwin, and Shparlinski, Igor. On the Uniformity of Distribution of Congruential Generators over Elliptic Curves. Department of Computing, Macquarie University, NSW 2109, Australia; {eelmaha, igor}@isc.mq.edu.qu.