NIST Special Publication 800-90A
Revision 1

Recommendation for Random Number Generation Using Deterministic Random Bit Generators

Elaine Barker
John Kelsey

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.SP.800-90Ar1

COMPUTER SECURITY

National Institute of Standards and Technology
U.S. Department of Commerce

NIST Special Publication 800-90A
Revision 1

Recommendation for Random Number Generation Using Deterministic Random Bit Generators

Elaine Barker
John Kelsey
Computer Security Division
Information Technology Laboratory

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.SP.800-90Ar1

June 2015

U.S. Department of Commerce
Penny Pritzker, Secretary

National Institute of Standards and Technology
Willie May, Under Secretary of Commerce for Standards and Technology and Director

Authority

National Institute of Standards and Technology Special Publication 800-90A Revision 1
Natl. Inst. Stand. Technol. Spec. Publ. 800-90A Rev. 1, 109 pages (June 2015)
CODEN: NSPUE2

This publication is available free of charge from:
http://dx.doi.org/10.6028/NIST.SP.800-90Ar1

Comments may be provided to:

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
Email: rbg_comments@nist.gov

Reports on Computer Systems Technology

Abstract

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

Keywords

Deterministic random bit generator (DRBG); entropy; hash function; random number generator.

Acknowledgements

The National Institute of Standards and Technology (NIST) gratefully acknowledges and appreciates contributions by Mike Boyle and Mary Baish from NSA for assistance in the development of this Recommendation. NIST also thanks the many contributions by the public and private sectors.

Table of Contents

List of Figures

List of Tables

1 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].

A DRBG is based on a DRBG mechanism as specified in this Recommendation and includes a source of randomness. A DRBG mechanism uses an algorithm (i.e., a DRBG algorithm) that produces a sequence of bits from an initial value that is determined by a seed that is determined from the output of the randomness source. Once the seed is provided and the initial value is determined, the DRBG is said to be instantiated and may be used to produce output. 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 an assurance of randomness. If the seed is kept secret, and the algorithm is well designed, the bits output by the DRBG will be unpredictable, up to the instantiated security strength of the DRBG.

The security provided by an RBG that uses a DRBG mechanism is a system implementation issue; both the DRBG mechanism and its randomness source must be considered when determining whether the RBG is appropriate for use by consuming applications.

2 Conformance Testing

Conformance testing for implementations of this Recommendation will be conducted within the framework of the Cryptographic Module Validation Program (CMVP) and the Cryptographic Algorithm Validation Program (CAVP). The requirements of this Recommendation are indicated by the word "**shall**." Some of these requirements may be out-of-scope for CMVP or CAVP validation testing, and thus are the responsibility of entities using, implementing, installing or configuring applications that incorporate this Recommendation.

3 Scope

This Recommendation includes:

1. Requirements for the use of DRBG mechanisms,
2. Specifications for DRBG mechanisms that use hash functions and block ciphers,
3. Implementation issues, and
4. Assurance considerations.

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

[2] DRBGS have also been called Pseudorandom Number (or Bit) Generators.

This Recommendation specifies several 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, i.e., 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 Appendix C for a discussion of DRBG mechanism selection.

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

NIST Special Publication (SP) 800-90B [SP 800-90B] provides guidance on designing and validating entropy sources. SP 800-90C [SP 800-90C] provides guidance on the construction of an RBG from a randomness source and an **approved** DRBG mechanism from this document (i.e., SP 800-90A).

4 Terms and Definitions

Algorithm	A clearly specified mathematical process for computation; a set of rules that, if followed, will give a prescribed result.
Approved	FIPS-approved, NIST-Recommended and/or validated by the Cryptographic Algorithm Validation Program (CAVP).
Approved entropy source	An entropy source that has been validated as conforming to [SP 800-90B].
Backtracking Resistance	An RBG provides *backtracking resistance* relative to time *T* if it provides assurance that an adversary that has knowledge of the state of the RBG at some time(s) subsequent to time *T* (but incapable of performing work that matches the claimed security strength of the RBG) would be unable to distinguish between observations of *ideal random bitstrings* and (previously unseen) bitstrings that are output by the RBG at or prior to time *T*. In particular, an RBG whose design allows the adversary to "backtrack" from the initially-compromised RBG state(s) to obtain knowledge of prior RBG states and the corresponding outputs (including the RBG state and output at time *T*) would not provide backtracking resistance relative to time *T*. (Contrast with *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.
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 one 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 middleware) 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 plaintext to ciphertext and vice versa,

	2. The generation of keying material, or 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 includes a DRBG mechanism and (at least initially) has access to a randomness source. The DRBG produces a 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. Contrast with *NRBG*.
DRBG Mechanism	The portion of an RBG that includes the functions necessary to instantiate and uninstantiate the RBG, generate pseudorandom bits, (optionally) reseed the RBG and test the health of the the DRBG mechanism.
DRBG Mechanism Boundary	A conceptual boundary that is used to explain the operations of a DRBG mechanism and its interaction with and relation to other processes. (See *min-entropy*.)
Entropy	A measure of the disorder, randomness or variability in a closed system. Min-entropy is the measure used in this Recommendation.
Entropy Input	An input bitstring that provides an assessed minimum amount of unpredictability for a DRBG mechanism. (See *min-entropy*.)
Entropy Source	A combination of a noise source (e.g., thermal noise or hard drive seek times), health tests, and an optional conditioning component. The entropy source produces random bitstrings to be used by an RBG.
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; the symbol ⊕, defined as: $0 \oplus 0 = 0$ $\quad$ $1 \oplus 0 = 1$ $0 \oplus 1 = 1$ $\quad$ $1 \oplus 1 = 0$ Equivalent to binary addition without carry.
Fresh Entropy	A bitstring output from an entropy source, an NRBG or a DRBG that has access to a Live Entropy Source that is being used to provide prediction resistance.
Full Entropy	For the purposes of this Recommendation, a source of full-entropy bitstrings serves as a practical approximation to a

	source of ideal random bitstrings of the same length (see *ideal random sequence*).
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.
Ideal Random Bitstring	See *Ideal Random Sequence*.
Ideal Random Sequence	Each bit of an ideal random sequence is unpredictable and unbiased, with a value that is independent of the values of the other bits in the sequence. Prior to the observation of the sequence, the value of each bit is equally likely to be 0 or 1, and the probability that a particular bit will have a particular value is unaffected by knowledge of the values of any or all of the other bits. An ideal random sequence of *n* bits contains *n* bits of entropy.
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 implementation 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 a DRBG instantiation. This can include both secret and non-secret information. Compare to *working state*.
Key	See *Cryptographic Key*.
Live Entropy Source	An **approved** entropy source (see [SP 800-90B]) that can provide an RBG with bits having a specified amount of

	entropy immediately upon request or within an acceptable amount of time, as determined by the user or application relying upon that RBG.
Min-entropy	The *min-entropy* (in bits) of a random variable *X* is the largest value *m* having the property that each observation of *X* provides at least *m* bits of information (i.e., the min-entropy of *X* is the greatest lower bound for the information content of potential observations of *X*). The min-entropy of a random variable is a lower bound on its entropy. The precise formulation for min-entropy is $-(\log_2 \max p_i)$ for a discrete distribution having *n* possible outputs with probabilities $p_1,\ldots, p_n$. Min-entropy is often used as a worst-case measure of the unpredictability of a random variable. Also see [SP 800-90B].
Non-Deterministic Random Bit Generator (Non-deterministic RBG) (NRBG)	An RBG that always has access to an *entropy source* and (when working properly) produces output bitstrings that have *full entropy*. Often called a True Random Number (or Bit) Generator. (Contrast with a *deterministic random bit generator*).
Nonce	A time-varying value that has at most a negligible chance of repeating, e.g., a random value that is generated anew for each use, a timestamp, a sequence number, or some combination of these.
Personalization String	An optional string of bits that is combined with a secret entropy input and (possibly) a nonce to produce a seed.
Prediction Resistance	An RBG provides prediction resistance relative to time *T* if it provides assurance that an adversary with knowledge of the state of the RBG at some time(s) prior to *T* (but incapable of performing work that matches the claimed *security strength* of the RBG) would be unable to distinguish between observations of ideal random bitstrings and (previously unseen) bitstrings output by the RBG at or subsequent to time *T*. In particular, an RBG whose design allows the adversary to step forward from the initially compromised RBG state(s) to obtain knowledge of subsequent RBG states and the corresponding outputs (including the RBG state and output at time *T*) would not provide prediction resistance relative to time *T*. (Contrast with *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*.
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. An RBG is either a DRBG or an NRBG.
Randomness Source	A component of a DRBG (which consists of a DRBG mechanism and a randomness source) that outputs bitstrings that are used as entropy input by the DRBG mechanism. The randomness source can be an entropy source or an RBG.
Reseed	To acquire additional bits that will affect the internal state of the DRBG mechanism.
Secure Channel	A path for transferring data between two entities or components that ensures confidentiality, integrity and replay protection, as well as mutual authentication between the entities or components. The secure channel may be provided using **approved** cryptographic, physical or procedural methods, or a combination thereof. Sometimes called a trusted channel.
Security Strength	A number associated with the amount of work (that is, the number of operations of some sort) that is required to break a cryptographic algorithm or system in some way. In this Recommendation, the security strength is specified in bits and is a specific value from the set {112, 128, 192, 256}. If the security strength associated with an algorithm or system is S bits, then it is expected that (roughly) 2^S basic operations are required to break it.
Seed	Noun : A string of bits that is used as input to a DRBG mechanism. 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 acquire bits with sufficient entropy for the desired security strength. These bits will be used as input to a DRBG mechanism to determine a portion of the initial internal state.

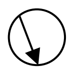
	Also see *reseed*.
Seedlife	The length of the seed period.
Seed Period	The period of time between instantiating or reseeding a DRBG with one seed and reseeding that DRBG with another seed.
Sequence	An ordered set of quantities.
Shall	Used to indicate a requirement of this Recommendation. "**Shall**" may be coupled with "not" to become "**shall not**."
Should	Used to indicate a highly desirable feature for a DRBG mechanism that is not necessarily required by this Recommendation. "**Should**" may be coupled with "not" to become "**should not**."
Source of Randomness	See *Randomness Source*.
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*.
Uninstantiate	The termination of a DRBG instantiation.
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 mechanism 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 Recommendation:

Abbreviation	Meaning
AES	Advanced Encryption Standard, as specified in [FIPS 197] .
DRBG	Deterministic Random Bit Generator.
FIPS	Federal Information Processing Standard.
HMAC	Keyed-Hash Message Authentication Code, as specified in [FIPS 198].
NIST	National Institute of Standards and Technology.
NRBG	Non-deterministic Random Bit Generator.
RBG	Random Bit Generator.
SP	NIST Special Publication.
TDEA	Triple Data Encryption Algorithm, as specified in [SP 800-67].

The following symbols are used in this Recommendation:

Symbol	Meaning
$+$	Addition.
$X \oplus Y$	Bitwise exclusive-or (also bitwise addition modulo 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 byte strings.
leftmost (*V*, *a*)	The leftmost a bits of V.
len (a)	The length in bits of string a.
min(a, b)	The minimum of a and b.
x **mod** n	The unique remainder r (where $0 \le r \le n$-1) when integer x is divided by n. For example, 23 mod 7 = 2.
rightmost (*V*, *a*)	The rightmost a bits of V.
select (V, a, b)	A substring of string V consisting of bit a through bit b.
	Used in a figure to illustrate a "switch" between input sources.
$\{a_1, ...a_i\}$	The internal state of the DRBG at a point in time. The types and number of the a_i values depends on the specific DRBG mechanism.

Symbol	Meaning
0x*ab*	Hexadecimal notation that is used to define a byte (i.e., eight bits) of information, where *a* and *b* each specify four bits of information and have values from the range {0, 1, 2,…F}. For example, 0xc6 is used to represent 11000110, where c is 1100, and 6 is 0110.
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 that uses a DRBG mechanism and discusses the major components of the DRBG mechanism.

— Section 8 provides concepts and general requirements for the implementation and use of a DRBG mechanism.

— Section 9 specifies the functions of a DRBG mechanism that were 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], and the block cipher algorithms specified in [FIPS 197] and [SP 800-67] (AES and TDEA, respectively).

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

This Recommendation also includes the following appendices:

— Appendix A provides conversion routines.

— Appendix B provides example pseudocode for each DRBG mechanism. Examples of the values computed for the DRBGs using each **approved** cryptographic algorithm and key size are available at http://csrc.nist.gov/groups/ST/toolkit/examples.html under the entries for SP 800-90A.

— Appendix C provides a discussion on DRBG mechanism selection.

— Appendix D provides references.

— Appendix E provides a list of modifications to SP 800-90A since it was first published.

7 Functional Model of a DRBG

Figure 1 provides a functional model of a DRBG (i.e., one type of RBG). A DRBG **shall** implement an **approved** DRBG mechanism from SP 800-90A and at least one **approved** randomness source (see Section 8.6.5), and may include additional optional sources, including sources for nonces, personalization strings, and additional input. The components of this model are discussed in the following subsections. DRBG constructions are also discussed in [SP 800-90C].

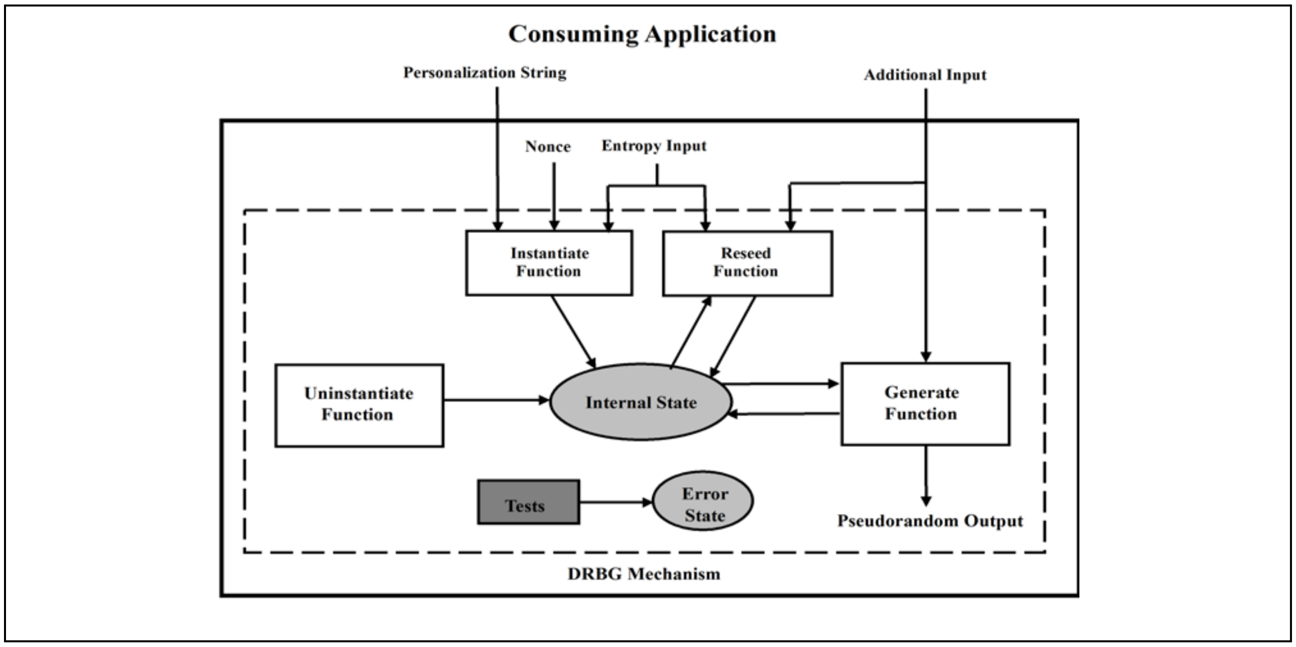


Figure 1: DRBG Functional Model

7.1 Entropy Input

Entropy input is provided to a DRBG mechanism for the seed (see Section 8.6) using a randomness source. 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 randomness source **shall** provide input that supports the security strength requested by the DRBG mechanism. Appropriate randomness sources are discussed in Section 8.6.5.

Ideally, the entropy input would have full entropy; however, the DRBG mechanisms have been specified so that input with full entropy is not required. This is accommodated by allowing the length of the entropy input to be longer than the required entropy (expressed in bits), as long as the total entropy meets the requirements of the DRBG mechanism. The entropy input can be defined to be of variable length (within specified limits), as well as fixed length. In all cases, the DRBG mechanism expects that when entropy input is requested, the returned bitstring will contain at least the requested amount of entropy. Additional entropy beyond the amount requested is not required, but is desirable.

7.2 Other Inputs

Other information may be obtained by a DRBG mechanism 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; for example, if time is used as an input, the format and reasonableness of the time could be checked. In most cases, a nonce is required during instantiation (see Sections 8.6.1 and 8.6.7). When required, the nonce is combined with the entropy input to create the initial DRBG seed.

A personalization string **should** be used during DRBG instantiation; when used, the personalization string is combined with the entropy input bits and possibly a nonce to create the initial DRBG seed. The personalization string **should** be unique for all instantiations of the same DRBG mechanism type (e.g., all instantiations of **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.

7.3 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 mechanism uses or acts upon. The internal state contains both administrative data (e.g., the security strength) and data that is acted upon and/or modified during the generation of pseudorandom bits (i.e., the working state).

7.4 The DRBG Mechanism Functions

The DRBG mechanism functions handle the DRBG's internal state. The DRBG mechanisms in this Recommendation have five separate functions:

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 possibly additional input; a new internal state for the next request is also generated.
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.
5. The health test function determines that the DRBG mechanism continues to function correctly.

8. DRBG Mechanism Concepts and General Requirements

8.1 DRBG Mechanism Functions

A DRBG mechanism requires instantiate, uninstantiate, generate, and health testing functions. A DRBG mechanism includes an optional 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 may be used to obtain pseudorandom bits for different purposes (e.g., DSA private keys and AES keys) and may be separately instantiated for each purpose, thus effectively creating two DRBGs.

A DRBG is instantiated using a seed and may 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 and end when the next seed is obtained or the DRBG is no longer used (see Figure 2).

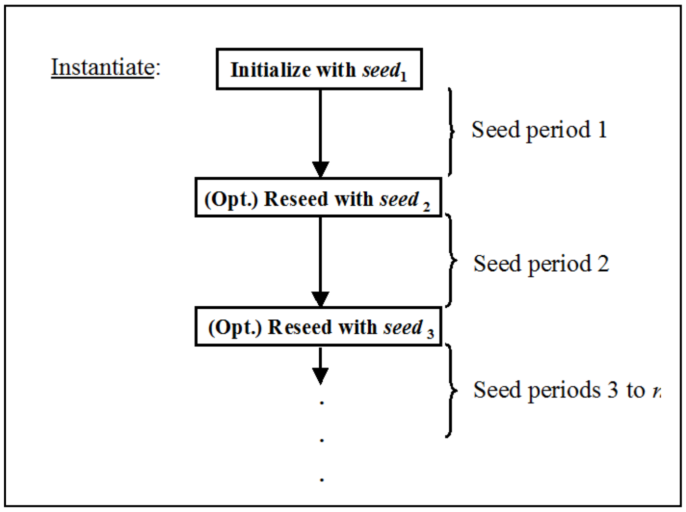


Figure 2: DRBG Instantiation

8.3 Internal States

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

1. The working state:
   a. One or more values that are derived from the seed and become part of the internal state; these values **shall** remain secret, and
   b. A count of the number of requests 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 mechanism implementation may 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.

8.4 Security Strengths Supported by an Instantiation

The DRBG mechanisms specified in this Recommendation support four security strengths: 112, 128, 192 or 256 bits. The security strength for the instantiation is requested during DRBG instantiation, and the instantiate function obtains the appropriate amount of entropy for the requested security strength. Each DRBG mechanism has restrictions on the security strength it can support, based on its design (see Section 10).

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 could 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, instead, be instantiated to support only a 128-bit security strength if the additional security provided by the 256-bit security strength is not required (e.g., by requesting only 128 bits of entropy during instantiation, rather than 256 bits of entropy).

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

Following instantiation, a request can be made to the generate function for pseudorandom bits (see Section 9.3). The pseudorandom bits returned from a DRBG **shall not** be used for any application that requires a higher security strength than the DRBG is instantiated to support. The security strength provided in these returned bits is the minimum of the security strength supported by the DRBG and the length of the bit string returned, i.e.:

$$Security\_strength\_of\_output = \mathbf{min}(output\_length, DRBG\_security\_strength).$$

A concatenation of bit strings resulting from multiple calls to a DRBG will not provide a security strength for the concatenated string that is greater than the instantiated security strength of the DRBG. For example, two 128-bit output strings requested from a DRBG that supports a128-bit security strength cannot be concatenated to form a 256-bit string with a security strength of 256 bits. A more complete discussion of this issue is provided in [SP 800-90C].

For each generate request, the security strength to be provided for the bits is requested. Any security strength can be requested during a call to the generate function, 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. Assuming that the request is valid, the requested number of bits is returned.

When an instantiation is used for multiple purposes, the minimum security strength 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 Mechanism Boundaries

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

The DRBG mechanism boundary should not be confused with a cryptographic module boundary, as specified in [FIPS 140]; the relationship between a cryptographic module boundary and a DRBG boundary is mentioned below, but is more fully discussed in [SP 800-90C].

Within a DRBG mechanism boundary,

1. The DRBG internal state and the operation of the DRBG mechanism functions **shall** only be affected according to the DRBG mechanism specification.
2. The DRBG internal state **shall** exist solely within the DRBG mechanism boundary. The internal state **shall not** be accessible by non-DRBG functions or other instantiations of that DRBG 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 mechanism boundary, except as specified for the DRBG pseudorandom bit outputs.

Each DRBG mechanism includes one or more cryptographic primitives (i.e., a hash function or block cipher algorithm). Other applications may use the same cryptographic primitive, but the DRBG's internal state and the DRBG mechanism functions **shall not** be affected by these other applications. For example, a DRBG mechanism may use the same hash-function code as a digital-signature application.

A DRBG mechanism'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. As further discussed in [SP 800-90C], the DRBG mechanism boundary (in this case) is contained within a cryptographic module boundary.

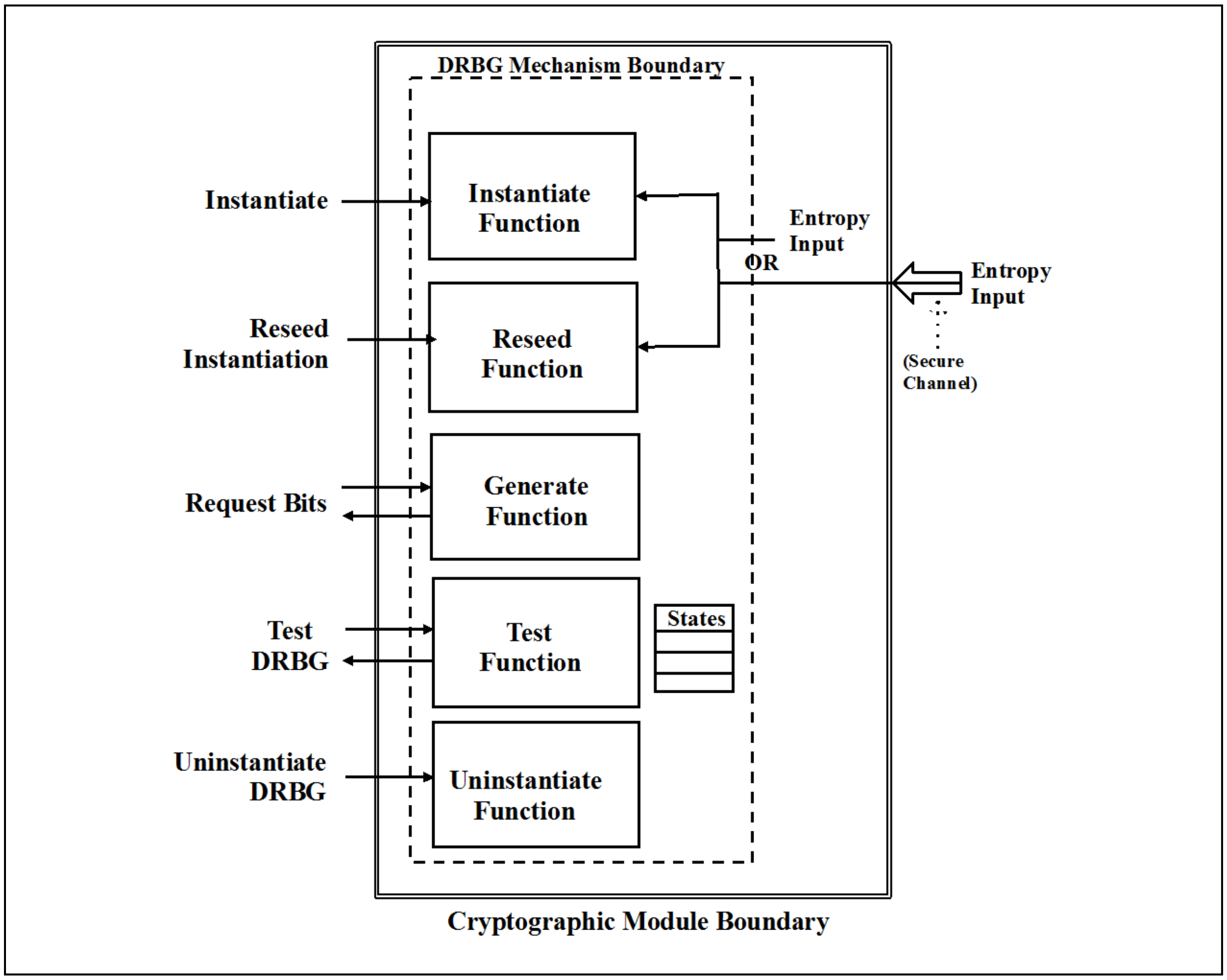


Figure 3: DRBG Mechanism Functions within a Single Device

Figure 4 provides an example of DRBG mechanism functions that are distributed across multiple devices. In this case, each device has a DRBG mechanism sub-boundary that contains the DRBG mechanism functions implemented on that device, and the DRBG mechanism sub-boundary is contained within a cryptographic module boundary, as is further discussed in [SP 800-90C]. The boundary around the entire DRBG mechanism includes the aggregation of sub-boundaries providing the DRBG mechanism functionality. Each sub-boundary may be contained within a different cryptographic module boundary, or multiple sub-boundaries may be contained within the same cryptographic module boundary.

The use of distributed DRBG-mechanism 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.

Each DRBG mechanism boundary or sub-boundary **shall** contain a health test function to test the “health” of other DRBG-mechanism functions within that boundary. In addition, a boundary or sub-boundary that contains an instantiate function **shall** contain an uninstantiate function in order to terminate an instantiation.

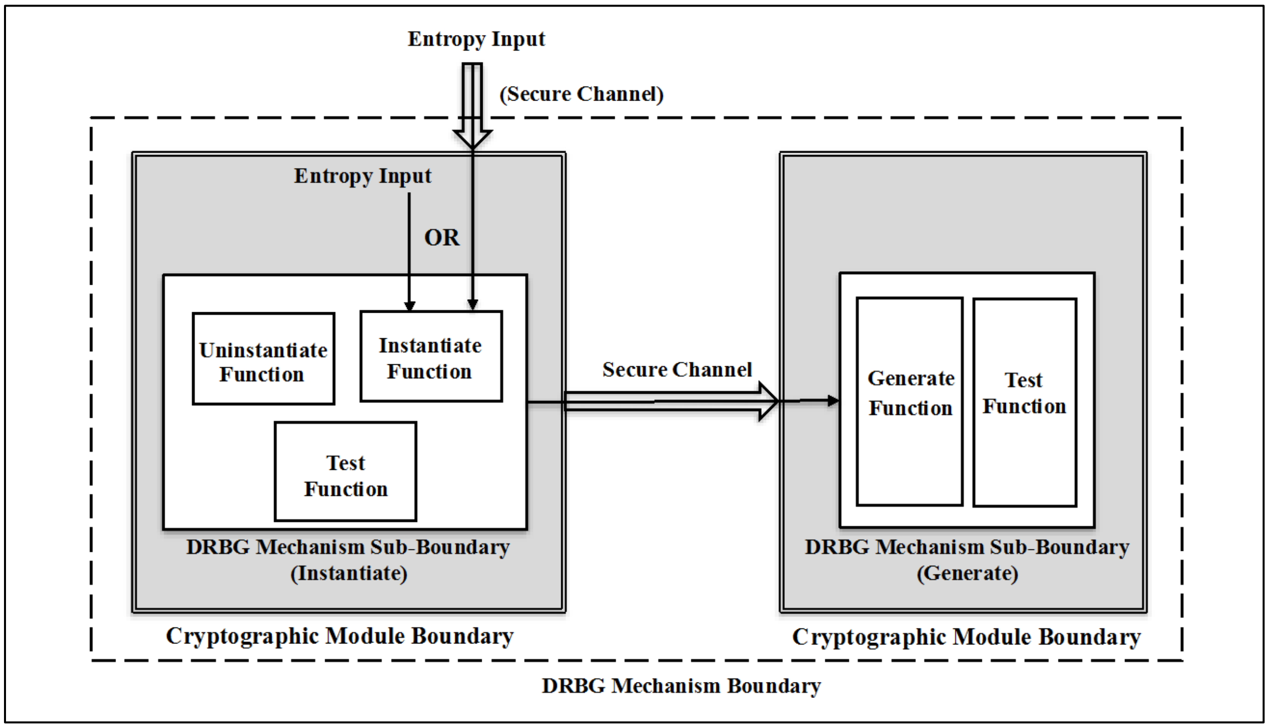


Figure 4: Distributed DRBG Mechanism Functions

When DRBG mechanism functions are distributed, a physically or cryptographically secure channel **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 mechanism sub-boundaries. The security provided by the secure channel **shall** be consistent with the security required by the consuming application. See Section 4 for a more complete definition of a secure channel.

For distributed DRBGs, each sub-boundary is the same as or is fully contained within a cryptographic module boundary.

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 that is used when calling the DRBG to obtain the first output bits.

Reseeding is a means of 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 mechanism **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 from a randomness source, a nonce and an optional (but recommended) 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, a nonce **shall** be used; requirements for the nonce are discussed in Section 8.6.7. A personalization string **should** also be used; requirements for the personalization string are discussed in Section 8.7.1.

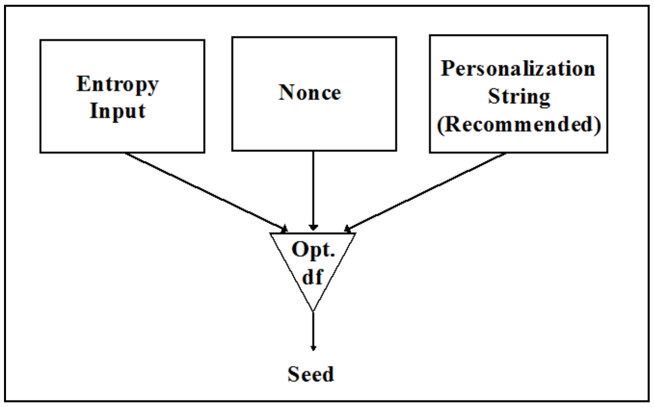


Figure 5: Seed Construction for Instantiation

Depending on the DRBG mechanism and the randomness source, a derivation function may be required to derive a seed from the seed material. However, in certain circumstances, the **CTR_DRBG** mechanism based on block cipher algorithms (see Section 10.2) may be implemented without a derivation function. When implemented in this manner, a (separate) 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 8.7.2. As in Section 8.6.1, a derivation function may be required for reseeding.

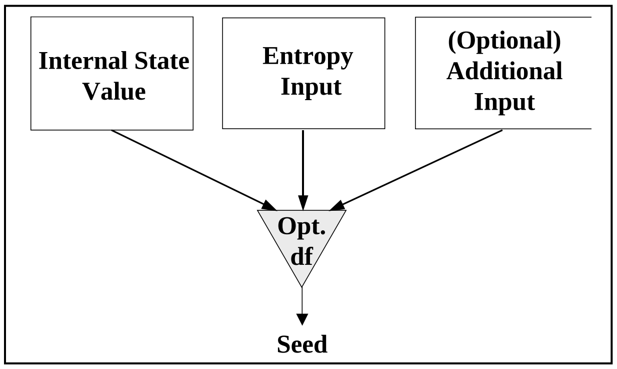


Figure 6: Seed Construction for Reseeding

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. Additional entropy may be provided in the nonce or the optional personalization string during instantiation, or in the additional input during reseeding and generation, but this is not required and does not increase the "official" security strength of the DRBG instantiation that is recorded in the internal state. The use of more entropy than the minimum value will offer a

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

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 mechanism and the security strength required by the consuming application, but **shall** be at least the number of bits of entropy required. See the tables in Section 10.

8.6.5 Randomness Source

A DRBG mechanism requires an **approved** randomness source during instantiation and reseeding, including whenever prediction resistance is requested (see Section 8.8). This input is requested using the **Get_entropy_input** function introduced in Section 9 and is specified in more detail in [SP 800-90C].

An **approved** randomness source is an entropy source that conforms to [SP 800-90B], or an RBG that conforms to [SP 800-90C] − either a DRBG or an NRBG.

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 protected at the same security strength as the keys.

The security of the DRBG depends on the secrecy of the entropy input. For this reason, the entropy input **shall** be treated as a critical security parameter (CSP) during cryptographic module validation. The entropy input for the DRBG function requiring the entropy input **shall** be obtained either from within the cryptographic module containing that function or from another cryptographic module and transported to the DRBG function's cryptographic module via a secure channel.

8.6.7 Nonce

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

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

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

Each nonce **shall** be unique to the cryptographic module in which instantiation is performed, but need not be secret. When used, the nonce **shall** be considered to be a critical security parameter.

A nonce may be composed of one (or more) of the following components (other components may also be appropriate):

1. A random value that is generated anew for each nonce, using an **approved** random bit generator.
2. A timestamp of sufficient resolution (detail) so that it is different each time it is used.
3. A monotonically increasing sequence number, or
4. A combination of a timestamp and a monotonically increasing sequence number, such that the sequence number is reset when and only when the timestamp changes. For example, a timestamp may show the date but not the time of day, so a sequence number is appended that will not repeat during a particular day.

For case 1 above, the random value could 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.

For case 2 above, the timestamp must be trusted. A trusted timestamp is generated and signed by an entity that is trusted to provide accurate time information.

The nonce provides greater assurance that the DRBG provides *security_strength* bits of security to the consuming application. If a DRBG were instantiated many times without a nonce, a compromise could become more likely. In some consuming applications, a single DRBG compromise could reveal long-term secrets (e.g., a compromise of the DSA per-message secret could reveal the signing key).

A nonce **shall** be generated within a cryptographic module boundary. This requirement does not preclude the generation of the nonce within a cryptographic module that is different from the cryptographic boundary containing the DRBG function with which the nonce is used (e.g., the cryptographic module boundary containing an instantiate function). However, in this scenario, there needs to be a secure channel to transport the nonce between the cryptographic-module boundaries. See the discussion of distributed DRBGs in Section 8.5 and distributed RBGs in [SP 800-90C].

8.6.8 Reseeding

Generating too many outputs from a seed (and other input information) may provide sufficient information for successfully predicting future outputs (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 have a finite seedlife (i.e., the number of outputs that are produced during a seed period); the maximum seedlife is dependent on the DRBG mechanism used. Implementations **shall** enforce the limits on seedlife specified for the DRBG mechanism used or more stringent limits selected by the implementer. When a DRBG's maximum seedlife is reached, the DRBG **shall not** generate outputs until it has been reseeded.

Reseeding is accomplished 1) by an explicit reseeding of the DRBG by the consuming application, 2) by the generate function when prediction resistance is requested (see Section 8.8) or 3) when the end of the seed life is determined during the generate function (see Section 9.3.1).

The reseeding of the DRBG **shall** be performed in accordance with the specification for a given DRBG mechanism. 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 would be to create an entirely new instantiation. However, reseeding is preferred over creating a new instantiation. If a DRBG instantiation was initially seeded with sufficient entropy, and the randomness source subsequently fails without being detected, then a new instantiation using the same (failed) source would not have sufficient entropy to operate securely. However, if there is an undetected failure in the randomness source of an already properly seeded DRBG instantiation, the DRBG instantiation will still retain any previous entropy when the reseed operation fails to introduce new entropy.

8.6.9 Seed Use

The seed that is used to initialize one instantiation of a DRBG **shall not** be intentionally used to reseed the same instantiation or used as the seed for another DRBG instantiation. In addition, a DRBG instantiation **shall not** reseed itself. 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 Entropy Input and Seed Separation

The seed used by a DRBG and the entropy input used to create that seed **shall not** intentionally be used for other purposes (e.g., domain parameter or prime number generation).

8.7 Other Input to the DRBG Mechanism

Other input may be provided during DRBG instantiation, 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 (see Section 8.7.2).

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 RBG.

8.7.1 Personalization String

A personalization string is an optional (but recommended) input to the instantiate function and is used to derive the seed (see Section 8.6.1). The personalization string may be obtained from inside or outside a cryptographic module, and may be an empty string. Note that a DRBG does not rely on a personalization string to provide entropy, even though entropy could be provided in the personalization string, and knowledge of the personalization string by an adversary does not degrade the security strength of a DRBG instantiation, as long as the entropy input is unknown. When used within a cryptographic module, a personalization string is not considered to be a critical security parameter.

The personalization string may contain secret information, but **shall not** include secret information that requires protection at a higher security strength than the DRBG being instantiated will support. For example, a personalization string to be used to instantiate a DRBG at 112 bits of security strength **shall not** include information requiring 128 bits of protection. A given implementation of a DRBG may support the use of a personalization string, but is not required to do so.

The intent of a personalization string is to introduce additional input into the instantiation of a DRBG. This personalization string might contain values unknown to an attacker, or values that tend to differentiate this DRBG instantiation from all others. Ideally, a personalization string will be set to some bitstring that is as unique as possible. Good sources for the personalization string contents include:

- Application identifiers,
- Device serial numbers,
- User identification,
- Per-module or per-device values,
- Timestamps,
- Network addresses,
- Special key values for this specific DRBG instantiation,
- Protocol version identifiers,
- Random numbers,
- Nonces, and
- Outputs from other approved or non-approved random bit generators.

8.7.2 Additional Input

Additional input may optionally be provided to the reseed and generate functions during requests. The additional input may be obtained from inside or outside a cryptographic module, and may include secret or public information. Note that a DRBG does not rely on additional input to provide entropy, even though entropy could be provided in the additional input, and knowledge of the additional input by an adversary does not degrade the security strength of a DRBG. However, if the additional input contains secret/private information (e.g., a social security number), that information **shall not** require protection at a higher security strength than the security strength supported by the DRBG. A given implementation of a DRBG may include the additional input, but is not required to do so. When used within a cryptographic module, the additional input used in DRBG requests is not considered to be a critical security parameter unless any secret information included in the additional input qualifies as a critical security parameter.

Additional input is optional for both the DRBG and the consuming application, and the ability to enter additional input may or may not be included in an implementation. The value of the 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 mechanism. 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 entropy input, the seed or one or more DRBG internal states.

8.8 Prediction Resistance and Backtracking Resistance

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 non-secret information.

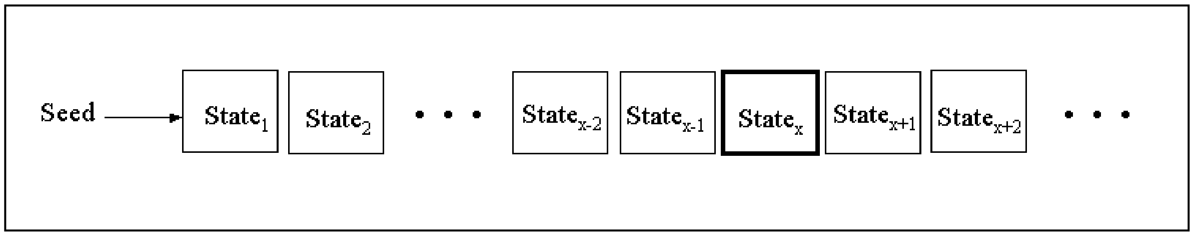


Figure 7: Sequence of DRBG States

<u>Backtracking Resistance</u>: Backtracking resistance is provided relative to time *T* if there is assurance that an adversary who has knowledge of the internal state of the DRBG at some time subsequent to time *T* would be unable to distinguish between observations of ideal random bitstrings and (previously unseen) bitstrings that were output by the DRBG prior to time *T*. This assumes that the adversary is incapable of performing the work required to negate the claimed security strength of the DRBG. 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 of the prior output sequence cannot distinguish it from random output with less work than is associated with the security strength of the instantiation; if the adversary knows only part of the prior output, he cannot determine any bit of that prior output sequence that he has not already seen with better than a 50-50 chance.

For example, suppose that an adversary knows *State*$_x$. Backtracking resistance means that:

a. The output bits from *State*$_1$ to *State*$_{x-1}$ cannot be distinguished from random output, and
b. The prior internal state values themselves (*State*$_1$ to *State*$_{x-1}$) cannot be recovered, given knowledge of the secret information in *State*$_x$.

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

<u>Prediction Resistance</u>: 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 output with less work than is associated with the security strength of the instantiation; if the adversary knows only part of the future output sequence, he cannot predict any bit of that future output sequence that he does not already know (with better than a 50-50 chance).

For example, suppose that an adversary knows *State*$_x$. Prediction resistance means that:

a. The output bits from *State*$_{x+1}$ and forward cannot be distinguished from an ideal random bitstring by the adversary, and

b. The future internal state values themselves ($State_{x+1}$ and forward) cannot be predicted (with better than a 50-50 chance), given knowledge of $State_x$.

Prediction resistance is provided relative to time *T* if there is assurance that an adversary with knowledge of the state of the RBG at some time(s) prior to *T* (but incapable of performing work that matches the claimed *security strength* of the RBG) would be unable to distinguish between observations of *ideal random bitstrings* and (previously unseen) bitstrings output by the RBG at or subsequent to time *T*. In particular, an RBG whose design allows the adversary to step forward from the initially compromised RBG state(s) to obtain knowledge of subsequent RBG states and the corresponding outputs (including the RBG state and output at time *T*) would not provide prediction resistance relative to time *T*.

Prediction resistance can be provided only by ensuring that a DRBG is effectively reseeded with fresh entropy between producing output for consecutive DRBG requests. That is, an amount of entropy that is sufficient to support the security strength of the DRBG being reseeded (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 DRBG internal state does not allow an adversary any useful knowledge about future DRBG internal states or outputs. Prediction resistance can be provided when the randomness source is or has direct or indirect access to an entropy source or an NRBG (see Section 8.6.5).

For example, suppose that an adversary knows internal $State_{x\text{-}2}$ (see Figure 7). If the adversary also knows the DRBG mechanism used, he then has enough information to compute $State_{x\text{-}1}$ and $State_x$. If prediction is then requested for the next bits that are to be output from the DRBG, new entropy bits will be inserted into the DRBG instantiation before $State_{x+1}$ is produced that will create a separation between $State_x$ and $State_{x+1}$, i.e., the adversary will not be able to compute $State_{x+1}$, simply by knowing $State_x$; the work required will be greatly increased by the entropy inserted during the prediction request.

The introduction of fresh entropy via reseeding will also make the DRBG less susceptible to cryptanalytic attack. **Whenever an entropy source is available, it is strongly recommended that DRBGs be requested to provide prediction resistance as often as is practical.**

9 DRBG Mechanism Functions

All DRBG mechanisms and algorithms are described in this document in pseudocode, which is intended to explain functionality. The pseudocode is not intended to constrain real-world implementations.

Except for the health test function, which is discussed in Section 11.3, the functions of the DRBG mechanisms in this Recommendation are specified as an algorithm and an "envelope" of pseudocode around that algorithm. The pseudocode in each envelope (provided in this section) checks the input parameters, obtains input not provided via the input parameters, accesses the appropriate DRBG algorithm and manages the internal state. A function need not be implemented using such envelopes, but the function **shall** have equivalent functionality.

During instantiation and reseeding (see Sections 9.1 and 9.2), entropy input and (usually) a nonce are acquired for constructing a seed as discussed in Sections 8.6.1 and 8.6.2. In the specifications of this Recommendation, a **Get_entropy_input** function is used for this purpose. The entropy input and nonce **shall** be provided as discussed in Sections 8.6.5 and 8.6.7 and in [SP 800-90C].

The **Get_entropy_input** function is specified in pseudocode in [SP 800-90C] for various RBG constructions; however, in general, the function 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*, *prediction_resistance_request*),
>
> 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. The *prediction_resistance_request* parameter indicates whether or not prediction resistance is to be provided during the request (i.e., whether fresh entropy is required[4]). A *status* code is returned from the function.

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

In the pseudocode in this section, two classes of error codes are returned: ERROR_FLAG and **CATASTROPHIC_ERROR_FLAG**. The handling of these classes of error codes is discussed in Section 11.4. The error codes may, in fact, provide information about the reason for the error; for example, when ERROR_FLAG is returned because of an incorrect input parameter, the ERROR_FLAG may indicate the problem.

Consuming applications **should** check the *status* returned from DRBG functions to determine whether or not the request was successful or if remediary action is required. For example, when

[4] Entropy input may be obtained from an entropy source or an NRBG, both of which provide fresh entropy. Entropy input could also be obtained from a DRBG that has access to an entropy source or NRBG.

The request for prediction resistance rules out the use of a DRBG that does not have access to either an entropy source or NRBG.

the instantiate function returns an error, an instantiation will not have been created, and an invalid *state_handle* will be returned (see Section 9.1); however, the lack of a *state_handle* will be detected in a subsequent reseed or generate request. When the reseed function returns an error (see Section 9.2), the indicated instantiation will not have been reseeded (i.e., the internal state will not have been injected with fresh entropy). When the generate function returns an error, a null string is returned as the output string (see Section 9.3.1) and **shall not** be used as pseudorandom output.

Comments are often included in the pseudocode in this Recommendation. A comment placed on a line that includes pseudocode applies to that line; a comment placed on a line containing no pseudocode applies to one or more lines of pseudocode immediately below that comment.

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. Obtains entropy input with entropy sufficient to support the security strength,
4. Obtains the nonce (if required),
5. Determines the initial internal state using the instantiate algorithm, and
6. If an implementation supports multiple simultaneous instantiations of the same DRBG, a *state_handle* for the internal state is returned to the consuming application (see below).

Let *working_state* be the working state for the particular DRBG mechanism (e.g., **HMAC_DRBG**), and let *min_length*, *max_ length*, and *highest_supported_security_strength* be defined for each DRBG mechanism (see Section 10). Let **Instantiate_algorithm** be a call to the appropriate instantiate algorithm for the DRBG mechanism (see Section 10).

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

Instantiate_function (*requested_instantiation_security_strength*, *prediction_resistance_flag, personalization_string*):

1. *requested_instantiation_security_strength*: A requested security strength for the instantiation. Implementations that support only one security strength do not require this parameter; however, any consuming application using that implementation must be aware of the security strength that is supported.

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. Implementations that always provide or do not support prediction resistance may not need to support this parameter if the intent is implicitly known. 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 an implementation. If the *prediction_resistance_flag* is not needed (i.e., it is known that prediction resistance is always performed or is not supported), then the *prediction_resistance_flag* input

parameter and instantiate process step 2 are omitted, and the *prediction_resistance_flag* is omitted from the internal state in step 11 of the instantiate process. In addition, step 6 can be modified to not perform a check for the *prediction_resistance_flag* when the flag is not used in an implementation; in this case, the **Get_entropy_input** call need not include the *prediction_resistance_request* parameter.

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 mechanism (see Section 10). If the input of a personalization string is not supported, then the *personalization_string* input parameter and step 3 of the instantiate process are omitted, and instantiate process step 9 is modified to omit the personalization string.

Required information not provided by the consuming application during instantiation (This information **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 mechanism (see Section 10).

2. *nonce*: A nonce as specified in Section 8.6.7. Note that if a random value is used in the nonce, the *entropy_input* and random portion of the *nonce* could be acquired using a single **Get_entropy_input** call (see step 6 of the instantiate process); in this case, the first parameter of the **Get_entropy_input** call is adjusted to include the entropy for the *nonce* (i.e., the *security_strength* is increased by at least ½ *security_strength*, and *min-length* is increased to accommodate the length of the nonce), instantiate process step 8 is omitted, and the *nonce* is omitted from the parameter list in instantiate process step 9.

   Note that in some cases, a nonce will not be used by a DRBG mechanism; in this case, step 8 is omitted, and the *nonce* is omitted from the parameter list in instantiate process step 9.

Output to a consuming application after instantiation:

1. *status*: The status returned from the instantiate function. If any status other than SUCCESS is returned, either no *state_handle* or an invalid *state_handle* **shall** be returned to the consuming application. 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 health test functions.

   If a state handle is not required for an implementation because the implementation does not support multiple simultaneous instantiations, a *state_handle* need not be returned. In this case, instantiate process step 10 is omitted, process step 11 is revised to save the only internal state, and process step 12 is altered to omit the *state_handle*.

Information retained within the DRBG mechanism boundary after instantiation:

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

Instantiate Process:

Comment: Check the validity of the input parameters.

1. If *requested_instantiation_security_strength* > *highest_supported_security_strength*, then return (ERROR_FLAG**,** Invalid).
2. If *prediction_resistance_flag* is set, and prediction resistance is not supported, then return (ERROR_FLAG, Invalid**)**.
3. If the length of the *personalization_string* > *max_personalization_string_length*, return (ERROR_FLAG, Invalid**)**.
4. Set *security_strength* to the lowest security strength greater than or equal to *requested_instantiation_security_strength* from the set {112, 128, 192, 256}.
5. Null step. Comment: This null step replaces a step from the original version of SP 800-90 without changing the step numbers.

   Comment: Obtain the entropy input.
6. (*status*, *entropy_input*) = **Get_entropy_input** (*security_strength*, *min_length*, *max_length*, *prediction_resistance_request*).

   Comment: *status* indications other than SUCCESS could be ERROR_FLAG or **CATASTROPHIC_ERROR_FLAG**, in which case, the status is returned to the consuming application to handle. The **Get_entropy_input** call could return a *status* of ERROR_FLAG to indicate that entropy is currently unavailable, and could return **CATASTROPHIC_ERROR_FLAG** to indicate that an entropy source failed.
7. If (*status* ≠ SUCCESS), return (*status*, Invalid).
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*, *security_strength*).
10. Get a *state_handle* for a currently empty internal state. If an empty internal state cannot be found, return (ERROR_FLAG, Invalid**)**.

11. Set the internal state for the new instantiation (e.g., as 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 administrative information to the appropriate values (e.g., the values of *security_strength* and the *prediction_resistance_flag*).
12. Return (SUCCESS, *state_handle*).

9.2 Reseeding a DRBG Instantiation

The reseeding of an instantiation is not required, but is recommended whenever a consuming 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 entropy is available).

The reseed function:

1. Checks the validity of the input parameters,
2. Obtains entropy input from a randomness source that supports the security strength of the DRBG, 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 instantiation (e.g., **HMAC_DRBG**) , let *min_length* and *max_ length* be defined for each DRBG mechanism, and let **Reseed_algorithm** be a call to the appropriate reseed algorithm for the DRBG mechanism (see Section 10).

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

Reseed_function (*state_handle, prediction_resistance_request, additional_input*):

1) *state_handle*: A pointer or index that indicates the internal state to be reseeded. If a state handle is not used by an implementation because the implementation does not support multiple simultaneous instantiations, a *state_handle* is not provided as input. Since there is only a single internal state in this case, reseed process step 1 obtains the contents of the internal state, and reseed process step 6 replaces the *working_state* of this internal state.

2) *prediction_resistance_request*: Indicates whether or not prediction resistance is to be provided during the request (i.e., whether or not fresh entropy bits are required)[5]. Without the explicit prediction resistance request, the entropy input could be provided from either a DRBG with no access to an entropy source (i.e., fresh entropy would not be provided), or the entropy input could be provided by an entropy source or by an RBG with access to an entropy source (i.e., fresh entropy would be provided in these cases).

   DRBGs that are implemented to always support prediction resistance or to never support prediction resistance do not require this parameter. However, when prediction resistance is not supported, 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 not supported, then the *prediction_resistance_request* input parameter and step 2 of the reseed process is omitted, and reseed process step 4 is modified to omit the *prediction_resistance_request* parameter.

   If prediction resistance is always performed, then the *prediction_resistance_request* input parameter and reseed process step 2 may be omitted, and reseed process step 4 is replaced by:

   (*status*, *entropy_input*) = **Get_entropy_input** (*security_strength*, *min_length*, *max_length*)

3) *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 mechanism (see Section 10). If the input by a consuming application of *additional_input* is not supported, then the input parameter and step 2 of the reseed process are omitted, and step 5 of the reseed process is modified to remove the *additional_input* from the parameter list.

Required information not provided by the consuming application during reseeding (This information **shall not** be provided by the consuming application as an input parameter during the reseed request):

1. *entropy_input*: Input bits containing entropy. This input **shall not** be provided by the DRBG instantiation being reseeded. 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 mechanism (see Section 10).

2. Internal state values required by the DRBG for the *working_state* and administrative information, as appropriate.

Output to a consuming application after reseeding:

1. *status*: The status returned from the function.

[5] A DRBG may be reseeded by an entropy source or an NRBG, both of which provide fresh entropy. A DRBG could also be reseeded by a DRBG that has access to an entropy source or NRBG. The request for prediction resistance during reseeding rules out the use of a DRBG that does not have access to either an entropy source or NRBG. See [SP 800-90C] for further discussion.

Information retained within the DRBG mechanism 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 (ERROR_FLAG**)**.
2. If *prediction_resistance_request* is set, and *prediction_resistance_flag* is not set, then return (ERROR_FLAG**)**.
3. If the length of the *additional_input* > *max_additional_input_length*, return (ERROR_FLAG**)**.

Comment: Obtain the entropy input.

4. (*status*, *entropy_input*) = **Get_entropy_input** (*security_strength*, *min_length*, *max_length*, *prediction_resistance_request*).

Comment: *status* indications other than SUCCESS could be ERROR_FLAG or **CATASTROPHIC_ERROR_FLAG**, in which case, the status is returned to the consuming application to handle. The **Get_entropy_input** call could return a *status* of ERROR_FLAG to indicate that entropy is currently unavailable, and could return **CATASTROPHIC_ERROR_FLAG** to indicate that an entropy source failed.

5. If (*status* ≠ SUCCESS), return (*status*).

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

6. *new_working_state* = **Reseed_algorithm** (*working_state*, *entropy_input*, *additional_input*).
7 Replace the *working_state* in the internal state for the DRBG instantiation (e.g., as indicated by *state_handle*) with the values of *new_working_state* obtained in step 6.
8. 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 application.

9.3.1 The Generate Function

Let *outlen* be the length of the output block of the cryptographic primitive (see Section 10). Let **Generate_algorithm** be a call to the appropriate generate algorithm for the DRBG mechanism (see Section 10), and let **Reseed_function** be a call to the reseed function in Section 9.2.

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

Generate_function (*state_handle, requested_number_of_bits, requested_security_strength***,** *prediction_resistance_request, additional_input*):

1. *state_handle*: A pointer or index that indicates the internal state to be used. If a state handle is not used by an implementation because the implementation does not support multiple simultaneous instantiations, a *state_handle* is not provided as input. The *state_handle* is then omitted from the input parameter list in process step 7.1, generate process steps 1 and 7.3 are used to obtain the contents of the internal state, and process step 10 replaces the *working_state* of this internal state.
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 mechanism.
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 the supported security strength.
4. *prediction_resistance_request*: Indicates whether or not prediction resistance is to be provided during the request. DRBGs that are implemented to always provide prediction resistance or that do not support prediction resistance do not require this parameter. However, when prediction resistance is not supported, 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 not supported, then the *prediction_resistance_request* input parameter and steps 5 and 9.2 of the generate process are omitted, and generate process steps 7 and 7.1 are modified to omit the check for the *prediction_resistance_request* term.

   If prediction resistance is always performed, then the *prediction_resistance_request* input parameter and generate process steps 5 and 9.2 may be omitted, and generate process steps 7 and 8 may be replaced by:

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

Comment: *status* indications other than SUCCESS could be ERROR_FLAG or **CATASTROPHIC_ERROR_FLAG**, in which case, the status is returned to the consuming application to handle. The **Get_entropy_input** call could return a *status* of ERROR_FLAG to indicate that entropy is currently unavailable, and could return **CATASTROPHIC_ERROR_FLAG** to indicate that an entropy source failed.

If (*status* ≠ SUCCESS), return (*status*, *Null*).

Using *state_handle*, obtain the new internal state.

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

Note that if the input of *additional_input* is not supported, then the *additional_input* parameter in the **Reseed_function** 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 mechanism (see Section 10). If the input of *additional_input* is not supported, then the input parameter, generate process steps 4 and 7.4, and the *additional_input* input parameter in generate process steps 7.1 and 8 are omitted.

Required information not provided by the consuming application during generation:

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

Output to a consuming application after generation:

1. *status*: The status returned from the generate function. If any status other than SUCCESS is returned, a *Null* string **shall** be returned as the pseudorandom bits.
2. *pseudorandom_bits*: The pseudorandom bits that were requested or a *Null* string.

Information retained within the DRBG mechanism boundary after generation:

Replaced internal state values (i.e., the new *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 (ERROR_FLAG, *Null***)**.
2. If *requested_number_of_bits* > *max_number_of_bits_per_request*, then return (ERROR_FLAG, *Null***)**.
3. If *requested_security_strength* > the *security_strength* indicated in the internal state, then return (ERROR_FLAG, *Null***)**.

4. If the length of the *additional_input* > *max_additional_input_length*, then return (ERROR_FLAG, *Null*).

5. If *prediction_resistance_request* is set, and *prediction_resistance_flag* is not set, then return (ERROR_FLAG, *Null*).

6. Clear the *reseed_required_flag*. Comment: See Section 9.3.2 for a discussion.

   Comment: Reseed if necessary (see Section 9.2).

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

   7.1 *status* = **Reseed_function** (*state_handle*, *prediction_resistance_request*, *additional_input*).

   Comment: *status* indications other than SUCCESS could be ERROR_FLAG or **CATASTROPHIC_ERROR_FLAG**, in which case, the status is returned to the consuming application to handle. The **Get_entropy_input** call could return a *status* of ERROR_FLAG to indicate that entropy is currently unavailable, and could return **CATASTROPHIC_ERROR_FLAG** to indicate that an entropy source failed.

   7.2 If (*status* ≠ SUCCESS), then return (*status*, *Null*).

   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 If the *prediction_resistance_flag* is set, then set the *prediction_resistance request* indication.

   9.3 Go to step 7.

10. Replace the old *working_state* in the internal state of the DRBG instantiation (e.g., as indicated by *state_handle*) with the values of *new_working_state*.

11. Return (SUCCESS, *pseudorandom_bits*).

Implementation notes:

If a reseed capability is not supported, or a reseed is not desired, then generate process steps 6 and 7 are removed; generate process 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_function** (*state_handle*).

   9.2 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 process (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, the *reseed_required_flag* is clear, so step 7 is 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 checks whether a reseed is required. If it is required, an appropriate *status* is 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. However, if the status indicates that a reseed is required (see step 9), then the *reseed_required_flag* is set, the *prediction_resistance_request* indicator is set if the instantiation is capable of performing prediction resistance, and processing continues by going back to step 7. This is intended to obtain fresh entropy for reseeding at the end of the reseed interval whenever access to fresh entropy is available (see the concept of Live Entropy Sources in [SP 800-90C]).

f. The substeps in step 7 are executed. The reseed function is called; any *additional_input* provided by the consuming application in the generate request is used during reseeding. 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 generate process step 7, resulting in a reseed, a new internal state for the instantiation, and setting the additional input to a

Null value; obtains pseudorandom bits (see generate process step 8); passes through generate process step 9, since another reseed will not be required; and continues with generate process step 10.

9.4 Removing a DRBG Instantiation

The internal state for an instantiation may need to be "released" by erasing (i.e., zeroizing) the contents of the internal state. The uninstantiate function:

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

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

Uninstantiate_function (*state_handle***):**

1. *state_handle*: A pointer or index that indicates the internal state to be "released". If a state handle is not used by an implementation because the implementation does not support multiple simultaneous instantiations, a *state_handle* is not provided as input. In this case, uninstantiate process step 1 is omitted, and process step 2 erases the internal state.

Output to a consuming application after uninstantiation:

1. *status*: The status returned from the function.

Information retained within the DRBG mechanism boundary after uninstantiation:

1. An empty internal state.

Uninstantiate Process:

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

10 DRBG Algorithm Specifications

Several DRBG mechanisms are specified in this Recommendation. The selection of a DRBG mechanism 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 **should** be conducted in order to select an appropriate DRBG mechanism. Conversion specifications required for the DRBG mechanism implementations (e.g., between integers and bitstrings) are provided in Appendix A. Pseudocode examples for each DRBG mechanism are provided in Appendix B. A detailed discussion on DRBG mechanism selection is provided in Appendix C.

Examples for determining correct implementation of each DRBG are available at http://csrc.nist.gov/groups/ST/toolkit/examples.html.

10.1 DRBG Mechanisms Based on Hash Functions

A DRBG mechanism may be based on a hash function that is one-way. The hash-based DRBG mechanisms 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 DRBG mechanisms 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 DRBG based on a hash function is the security strength of the hash function for pre-image resistance; these security strengths are provided in [SP 800-107]. [SP 800-57] identifies hash functions that can be used to support a required security strength.

This Recommendation supports only four security strengths: 112, 128, 192, and 256 bits. Table 2 specifies the values that **shall** be used for the function envelopes[6] and DRBG algorithm for each **approved** hash function.

[6] Discussed in Section 9.

Table 2: Definitions for Hash-Based DRBG Mechanisms

	SHA-1	**SHA-224 and SHA-512/224**	**SHA-256 and SHA-512/256**	**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*)	2^{35} bits				
Seed length (*seedlen*) for Hash_DRBG	440	440	440	888	888
Maximum personalization string length (*max_personalization_string_length*)	2^{35} bits				
Maximum additional_input length (*max_additional_input_length*)	2^{35} bits				
max_number_of_bits_per_request	2^{19} bits				
Maximum number of requests between reseeds (*reseed_interval*)	2^{48}				

Note that since SHA-224 is based on SHA-256, there is no efficiency benefit when using SHA-224, rather than SHA-256. Also note that since SHA-384, SHA-512/224 and SHA-512/256 are based on SHA-512, there is no efficiency benefit for using these three SHA mechanisms, rather than using SHA-512. However, efficiency is just one factor to consider when selecting the appropriate hash function to use as part of a DRBG mechanism.

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 throughout a **Hash_DRBG** instantiation. **Hash_DRBG** uses the derivation function specified in Section 10.3.1 during instantiation and reseeding. The security strength of the hash function to be used **shall** meet or exceed the desired security strength required by the consuming application for DRBG output (see [SP 800-57]).

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. 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 available for the DRBG instantiation.

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

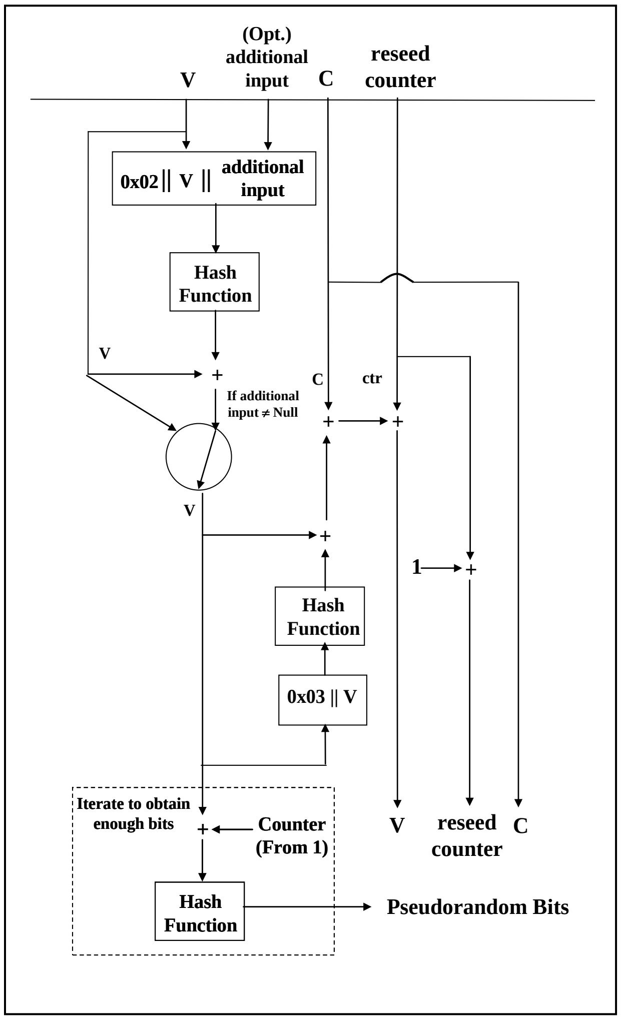


Figure 8: Hash_DRBG

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** specified in Section 9.1. Process step 9 of that function calls the instantiate algorithm in this section.

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.

The instantiate algorithm:

Let **Hash_df** be the hash derivation function specified in Section 10.3.1 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 mechanism (see step 9 of the instantiate process in Section 9.1).

Hash_DRBG_Instantiate_algorithm (*entropy_input*, *nonce*, *personalization_string*, *security_strength***):**

1. *entropy_input*: The string of bits obtained from the randomness 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. Note that the length of the *personalization_string* may be zero.
4. *security_strength*: The security strength for the instantiation. This parameter is optional for **Hash_DRBG**, since it is not used.

Output:

1. *initial_working_state*: The initial values for *V*, *C*, 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: Precede *V* with a byte of zeros.
5. *reseed_counter* = 1.
6. **Return** (*V*, *C*, *reseed_counter*).

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** specified in Section 9.2. Process step 6 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 10.3.1 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 mechanism (see step 6 of the reseed process in Section 9.2):

Hash_DRBG_Reseed_algorithm (*working_state*, *entropy_input*, *additional_input***):**

1. *working_state*: The current values for *V*, *C*, and *reseed_counter* (see Section 10.1.1.1).
2. *entropy_input*: The string of bits obtained from the randomness source.

3. *additional_input*: The additional input string received from the consuming application. Note that the length of the *additional_input* string may be zero.

Output:

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

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 specified in Section 9.3. 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 in Table 2 of Section 10.1. Note that for this DRBG mechanism, 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 mechanism (see step 8 of the generate process in Section 9.3):

Hash_DRBG_Generate_algorithm (*working_state, requested_number_of_bits, additional_input***):**

1. *working_state*: The current values for *V*, *C*, 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. Note that the length of the *additional_input* string may be zero.

Output:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS,** 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*, 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
   2.1 $w$ = Hash (0x02 || $V$ || additional_input).
   2.2 $V = (V + w) \bmod 2^{seedlen}$.
3. (returned_bits) = Hashgen (requested_number_of_bits, $V$).
4. $H$ = Hash (0x03 || $V$).
5. $V = (V + H + C + reseed\_counter) \bmod 2^{seedlen}$.
6. reseed_counter = reseed_counter + 1.
7. Return (SUCCESS, returned_bits, $V$, $C$, reseed_counter).
```

Hashgen (*requested_number_of_bits*, *V***)**:

Input:

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

Output:

1. *returned_bits*: The generated bits to be returned to the generate function.

Hashgen 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 $w$ = Hash (data).
   4.2 $W = W \| w$.
   4.3 $data = (data + 1) \bmod 2^{seedlen}$.
5. returned_bits = leftmost ($W$, requested_no_of_bits).
6. Return (returned_bits).
```

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 mechanism uses the **HMAC_DRBG_Update** function specified in Section 10.1.2.2 and the **HMAC** function within the **HMAC_DRBG_Update** function as the derivation function during instantiation and reseeding. The same hash function **shall** be used throughout an **HMAC_DRBG** instantiation. The hash function used **shall** meet or exceed the security strength required by the consuming application for DRBG output (see [SP 800-57]).

Figure 9 depicts the **HMAC_DRBG** in three stages. **HMAC_DRBG** is specified using an internal function (**HMAC_DRBG_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 **HMAC_DRBG_Update** function.

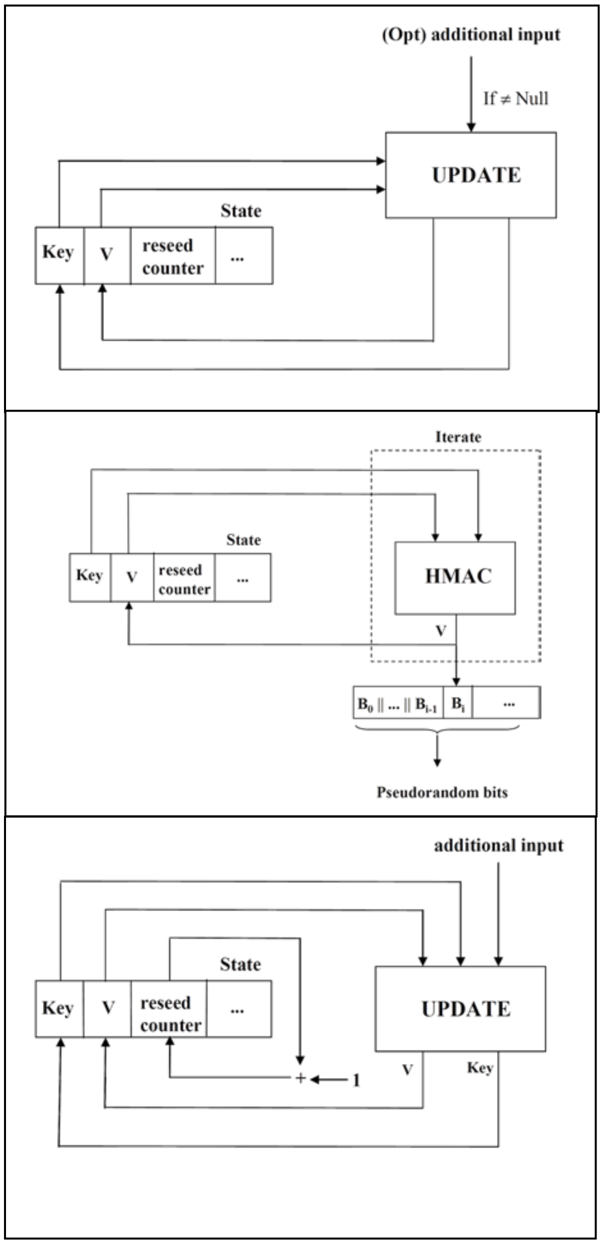


Figure 9: HMAC_DRBG Generate Function

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 *outlen*-bit *Key*, which is updated at least once each time that the DRBG mechanism 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 instantiation.

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

10.1.2.2 The HMAC_DRBG Update Function (Update)

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

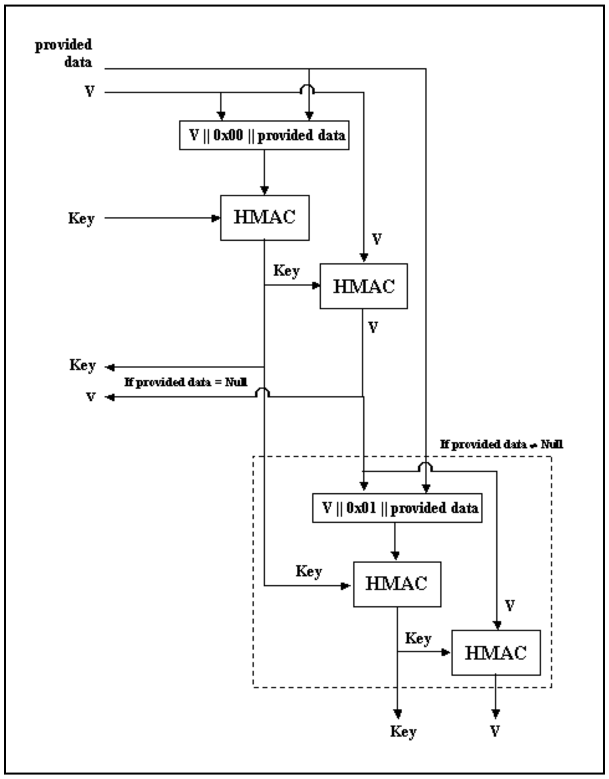


Figure 10: HMAC_DRBG_Update Function

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

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

HMAC_DRBG_Update (*provided_data, K, V***):**

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*, *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** specified in Section 9.1. Process step 9 of that function calls the instantiate algorithm specified in this section. 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 **HMAC_DRBG_Update** be the function specified in Section 10.1.2.2. The output 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 mechanism (see step 9 of the instantiate process in Section 9.1):

HMAC_DRBG_Instantiate_algorithm (*entropy_input*, *nonce*, *personalization_string*, *security_strength***):**

1. *entropy_input*: The string of bits obtained from the randomness 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. Note that the length of the *personalization_string* may be zero.
4. *security_strength*: The security strength for the instantiation. This parameter is optional for **HMAC_DRBG**, since it is not used.

Output:

1. *initial_working_state*: The initial 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*) = **HMAC_DRBG_Update** (*seed_material*, *Key*, *V*).
5. *reseed_counter* = 1.
6. Return (*V*, *Key*. *reseed_counter*).

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** specified in Section 9.2. Process step 6 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 **HMAC_DRBG_Update** be the function specified in Section 10.1.2.2. The following process or its equivalent **shall** be used as the reseed algorithm for this DRBG mechanism (see step 6 of the reseed process in Section 9.2):

HMAC_DRBG_Reseed_algorithm (*working_state, entropy_input, additional_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 randomness source.
3. *additional_input*: The additional input string received from the consuming application. Note that the length of the *additional_input* string may be zero.

Output:

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

HMAC_DRBG Reseed Process:

1. *seed_material* = *entropy_input* || *additional_input*.
2. (*Key*, *V*) = **HMAC_DRBG_Update** (*seed_material*, *Key*, *V*).
3. *reseed_counter* = 1.
4. **Return** (*V*, *Key*, *reseed_counter*).

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** specified in Section 9.3. 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 mechanism. 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 mechanism (see step 8 of the generate process in Section 9.3):

HMAC_DRBG_Generate_algorithm (*working_state, requested_number_of_bits, additional_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. Note that the length of the *additional_input* string may be zero.

Output:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS,** 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. If $additional\_input \neq Null$, then $(Key, V)$ =
   HMAC_DRBG_Update ($additional\_input, Key, V$).
3. $temp = Null$.
4. While (len ($temp$) < $requested\_number\_of\_bits$) do:
   4.1  $V$ = HMAC ($Key, V$).
   4.2  $temp = temp \| V$.
5. $returned\_bits$ = leftmost ($temp, requested\_number\_of\_bits$).
6. $(Key, V)$ = HMAC_DRBG_Update ($additional\_input, Key, V$).
7. $reseed\_counter = reseed\_counter + 1$.
8. Return (SUCCESS, $returned\_bits, Key, V, reseed\_counter$).
```

10.2 DRBG Mechanism Based on Block Ciphers

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

The maximum security strength that can be supported by the DRBG depends on the block cipher and key size used; the security strengths that can be supported by the block ciphers and key sizes are provided in [SP 800-57].

10.2.1 CTR_DRBG

CTR_DRBG uses an **approved** block cipher algorithm in the counter mode as specified in [SP 800-38A], but allows the counter field to be a subset of the input block, as specified in [SP 800-38D]. Note that for TDEA, the input and output block lengths are 64 bits, and for AES, the lengths are 128 bits.

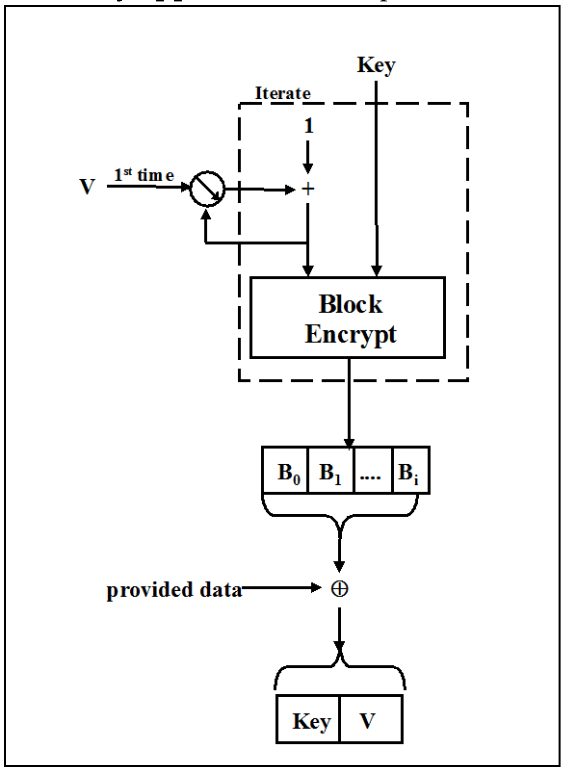


Figure 11: CTR_DRBG Update Function

The same block cipher algorithm and key length **shall** be used for all block cipher operations of this DRBG. 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 (**CTR_DRBG_Update**). Figure 11 depicts the **CTR_DRBG_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.

Table 3 specifies the values that **shall** be used for the function envelopes and **CTR_DRBG** mechanism.

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]			
Input and output block length (*blocklen*)	64	128	128	128
Counter field length (*ctr_len*)	$4 \le ctr\_len \le blocklen$			
Key length (*keylen*)	168	128	192	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*)	2^{35} bits			
Maximum personalization string length (*max_personalization_string_length*)	2^{35} bits			
Maximum additional_input length (*max_additional_input_length*)	2^{35} bits			
If a derivation function is not used:				
Minimum entropy input length (*min _length* = *blocklen* + *keylen*)	*seedlen*			
Maximum entropy input length (*max _length* = *blocklen* + *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 (for $B = (2^{ctr\_len} - 4) \times blocklen$)	**min**$(B, 2^{13})$	**min**$(B, 2^{19})$		
Maximum number of requests between reseeds (*reseed_interval*)	2^{32}	2^{48}		

Note that the claimed security strength for **CTR_DRBG** depends on limiting the total number of generate requests and the bits provided per generate request according to the table above.

Without these limits, it becomes possible, in principle, for an attacker to observe enough outputs from **CTR_DRBG** to distinguish its outputs from ideal random bits.

The **CTR_ DRBG** may be implemented to use the block cipher derivation function specified in Section 10.3.2 during instantiation and reseeding. However, the DRBG mechanism is specified to allow an implementation tradeoff with respect to the use of this derivation function. The use of the derivation function is optional if either an **approved** RBG or an entropy source provides full entropy output when entropy input is requested by the DRBG mechanism. Otherwise, the derivation function **shall** be used.

Table 3 provides the lengths required for the *entropy_input*, *personalization_string* and *additional_input* for each case.

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].

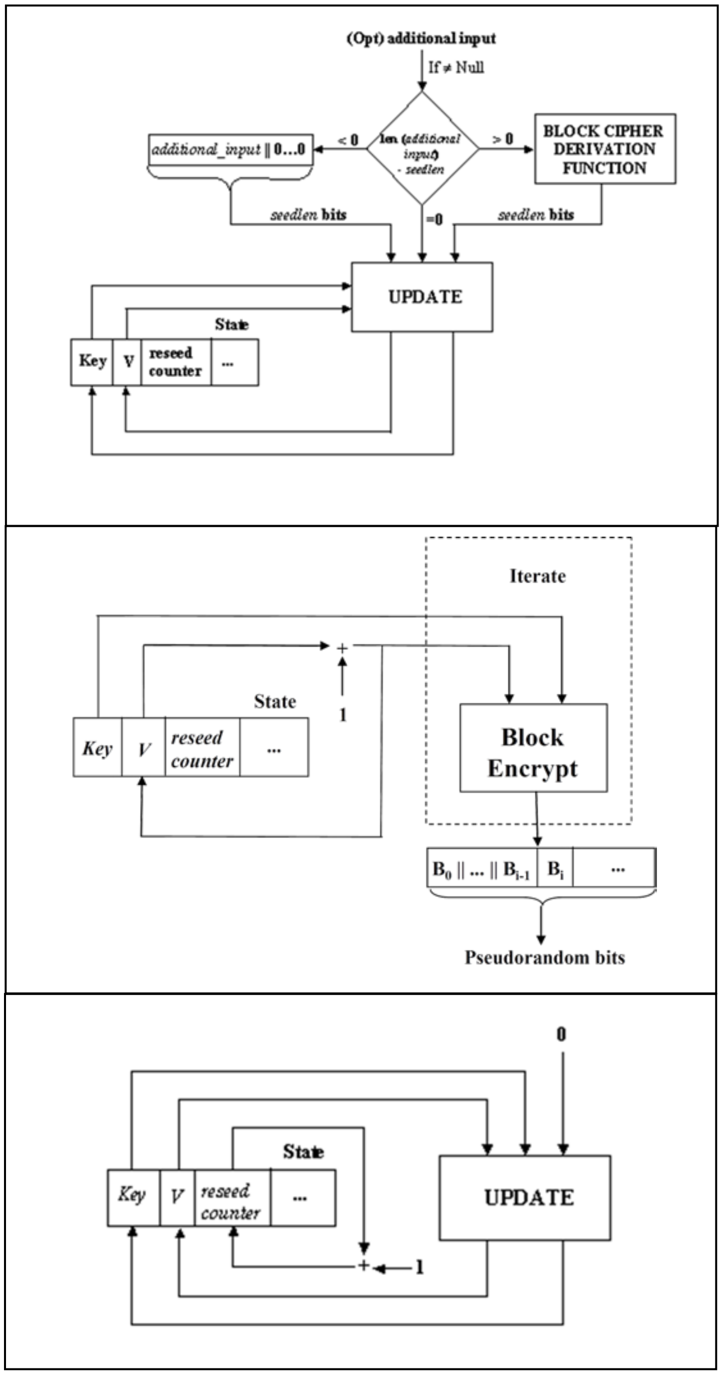


Figure 12: CTR-DRBG

10.2.1.1 CTR_DRBG Internal State

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

1. The *working_state*:
    a. The value *V* of *blocklen* bits, which is updated each time another *blocklen* bits of output are produced.
    b. The *keylen*-bit *Key*, which is updated whenever a predetermined number of output blocks are generated.
    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 instantiation.

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

10.2.1.2 The Update Function (CTR_DRBG_Update)

The **CTR_DRBG_Update** function updates the internal state of the **CTR_DRBG** using the *provided_data*. The values for *blocklen*, *keylen* and *seedlen* are provided in Table 3 of Section 10.2.1. The value of *ctr_len* is known by an implementation. The block cipher operation in step 2.2 of the **CTR_DRBG_UPDATE** process uses the selected block cipher algorithm. The specification of **Block_Encrypt** is discussed in Section 10.3.3.

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

CTR_DRBG_Update (*provided_data*, *Key*, *V*):

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 ($\mathbf{len}(temp) < seedlen$) do
   2.1 If $ctr\_len < blocklen$
       2.1.1 $inc = (\mathbf{rightmost}(V, ctr\_len) + 1) \bmod 2^{ctr\_len}$.
       2.1.2 $V = \mathbf{leftmost}(V, blocklen - ctr\_len) \,||\, inc$.
       Else $V = (V+1) \bmod 2^{blocklen}$.
   2.2 $output\_block = \mathbf{Block\_Encrypt}(Key, V)$.
   2.3 $temp = temp \,||\, output\_block$.
3. $temp = \mathbf{leftmost}(temp, seedlen)$.
4 $temp = temp \oplus provided\_data$.
5. $Key = \mathbf{leftmost}(temp, keylen)$.
6. $V = \mathbf{rightmost}(temp, blocklen)$.
```

7. Return (*Key*, *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** specified in Section 9.1. Process step 9 of that function calls the instantiate algorithm specified in this section. 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:

For this DRBG mechanism, there are two cases for processing. In each case, let **CTR_DRBG_Update** be the function specified in Section 10.2.1.2. The output block length (*blocklen*), key length (*keylen*), seed length (*seedlen*) and *security_strengths* for the block cipher algorithms are provided in Table 3 of Section 10.2.1.

10.2.1.3.1 Instantiation When a Derivation Function is Not Used

When instantiation is performed using this method, full-entropy input is required, and a nonce is not used. The following process or its equivalent **shall** be used as the instantiate algorithm for this DRBG mechanism:

CTR_DRBG_Instantiate_algorithm (*entropy_input*, *personalization_string*, *security_strength***):**

1. *entropy_input*: The string of bits obtained from the randomness source.
2. *personalization_string*: The personalization string received from the consuming application. Note that the length of the *personalization_string* may be zero.
3. *security_strength*: The security strength for the instantiation. This parameter is optional for **CTR_DRBG**.

Output:

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

CTR_DRBG Instantiate Process:

1. *temp* = **len** (*personalization_string*).

   Comment: Ensure that the length of the *personalization_string* is exactly *seedlen* bits. Note that in Section 9.1, processing step 3 obtained an *entropy_input* of *seedlen* bits using Table 3 to define the minimum and maximum lengths, which are both equal to *seedlen* bits.

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

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

4. *Key* = 0^{keylen}. Comment: *keylen* bits of zeros.
5. *V* = $0^{blocklen}$. Comment: *blocklen* bits of zeros.
6. (*Key*, *V*) = **CTR_DRBG_Update** (*seed_material*, *Key*, *V*).
7. *reseed_counter* = 1.
8. Return (*V*, *Key*, *reseed_counter*).

10.2.1.3.2 Instantiation When a Derivation Function is Used

When instantiation is performed using this method, the entropy input may or may not have full entropy; in either case, a nonce is required.

Let **df** be the derivation function specified in Section 10.3.2. When instantiation is performed using this method, a nonce is required, whereas using the method in Section 10.2.1.3.1 does not require a nonce, since full entropy is provided when using that method.

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

CTR_DRBG_Instantiate_algorithm (*entropy_input*, *nonce*, *personalization_string*, *security_strength***):**

1. *entropy_input*: The string of bits obtained from the randomness 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. Note that the length of the *personalization_string* may be zero.
4. *security_strength*: The security strength for the instantiation. This parameter is optional for **CTR_DRBG**, since it is not used.

Output:

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

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* = **df** (*seed_material*, *seedlen*).
3. *Key* = 0^{keylen}. Comment: *keylen* bits of zeros.
4. *V* = $0^{blocklen}$. Comment: *blocklen* bits of zeros.
5. (*Key*, *V*) = **CTR_DRBG_Update** (*seed_material*, *Key*, *V*).
6. *reseed_counter* = 1.
7. Return (*V*, *Key*, *reseed_counter*).

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** specified in Section 9.2. Process step 6 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:

For this DRBG mechanism, there are two cases for processing. In each case, let **CTR_DRBG_Update** be the function specified in Section 10.2.1.2. The seed length (*seedlen*) is provided in Table 3 of Section 10.2.1.

10.2.1.4.1 Reseeding When a Derivation Function is Not Used

When reseeding is performed using this method, full-entropy input is required.

The following process or its equivalent **shall** be used as the reseed algorithm for this DRBG mechanism (see step 6 of the reseed process in Section 9.2):

CTR_DRBG_Reseed_algorithm (*working_state*, *entropy_input*, *additional_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 randomness source.
3. *additional_input*: The additional input string received from the consuming application. Note that the length of the *additional_input* string may be zero.

Output:

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

CTR_DRBG Reseed Process:

```
1. $temp$ = 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. If ($temp < seedlen$), then $additional\_input = additional\_input \| 0^{seedlen - temp}$.
3. $seed\_material = entropy\_input \oplus additional\_input$.
4. ($Key$, $V$) = CTR_DRBG_Update ($seed\_material$, $Key$, $V$).
5. $reseed\_counter$ = 1.
6. Return ($V$, $Key$, $reseed\_counter$).
```

10.2.1.4.2 Reseeding When a Derivation Function is Used

Let **df** be the derivation function specified in Section 10.3.2.

The following process or its equivalent **shall** be used as the reseed algorithm for this DRBG mechanism (see reseed process step 6 of Section 9.2):

CTR_DRBG_Reseed_algorithm (*working_state*, *entropy_input*, *additional_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 randomness source.
3. *additional_input*: The additional input string received from the consuming application. Note that the length of the *additional_input* string may be zero.

Output:

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

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 = df (seed_material, seedlen).
3. (Key, V) = CTR_DRBG_Update (seed_material, Key, V).
4. reseed_counter = 1.
5. Return (V, Key, reseed_counter).
```

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** specified in Section 9.3. 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 *blocklen* are provided in Table 3 of Section 10.2.1. If the derivation function is not used, then the maximum allowed length of *additional_input* = *seedlen*.

For this DRBG mechanism, there are two cases for the processing. For each case, let **CTR_DRBG_Update** be the function specified in Section 10.2.1.2, and let **Block_Encrypt** be the function specified in Section 10.3.3. The seed length (*seedlen*) and the value of *reseed_interval* are provided in Table 3 of Section 10.2.1. The value of *ctr_len* is known by an implementation.

10.2.1.5.1 Generating Pseudorandom Bits When a Derivation Function is Not Used

This method of generating bits is used when a derivation function is not used by an implementation.

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

CTR_DRBG_Generate_algorithm (*working_state, requested_number_of_bits, additional_input***):**

1. *working_state*: The current values for *V*, *Key*, 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. Note that the length of the *additional_input* string may be zero.

Output:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS,** 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*, and *reseed_counter*.

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
                                   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, pad with zero bits.
       2.1  temp = len (additional_input).
       2.2  If (temp < seedlen), then
            additional_input = additional_input || $0^{seedlen - temp}$.
       2.3  (Key, V) = CTR_DRBG_Update (additional_input, Key, V).
   Else additional_input = $0^{seedlen}$.
3. temp = Null.
4. While (len (temp) < requested_number_of_bits) do:
   4.1  If ctr_len < blocklen
            4.1.1 inc = (rightmost (V, ctr_len) + 1) mod $2^{ctr\_len}$.
            4.1.2 V = leftmost (V, blocklen-ctr_len) || inc.
        Else V = (V+1) mod $2^{blocklen}$.
        4.2  output_block = Block_Encrypt (Key, V).
        4.3  temp = temp || output_block.
    5.  returned_bits = leftmost (temp, requested_number_of_bits).
                                 Comment: Update for backtracking resistance.
    6.  (Key, V) = CTR_DRBG_Update (additional_input, Key, V).
    7.  reseed_counter = reseed_counter + 1.
    8.  Return (SUCCESS, returned_bits, Key, V, reseed_counter).
```

10.2.1.5.2 Generating Pseudorandom Bits When a Derivation Function is Used

This method of generating bits is used when a derivation function is used by an implementation.

Let **df** be the derivation function specified in Section 10.3.2.

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

CTR_DRBG_Generate_algorithm (*working_state*, *requested_number_of_bits*, *additional_input***):**

1. *working_state*: The current values for *V*, *Key*, 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. Note that the length of the *additional_input* string may be zero.

Output:

1. *status*: The status returned from the function. The *status* will indicate **SUCCESS,** 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*, and *reseed_counter*.

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) = CTR_DRBG_Update (additional_input, Key, V).
        Else additional_input = $0^{seedlen}$.
    3.  temp = Null.
```

```
4. While (len (temp) < requested_number_of_bits) do:
    4.1 If ctr_len < blocklen
        4.1.1 inc = (rightmost (V, ctr_len) + 1) mod $2^{ctr\_len}$.
        4.1.2 V = leftmost (V, blocklen-ctr_len) || inc.
        Else
        4.1.2 V = (V+1) mod $2^{blocklen}$.
    4.2 output_block = Block_Encrypt (Key, V).
    4.3 temp = temp || output_block.
5. returned_bits = leftmost (temp, requested_number_of_bits).
                                        Comment: Update for backtracking resistance.
6. (Key, V) = CTR_DRBG_Update (additional_input, Key, V).
7. reseed_counter = reseed_counter + 1.
8. Return (SUCCESS, returned_bits, Key, V, reseed_counter).
```

10.3 Auxiliary 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 the use of hash functions (see Section 10.3.1), and the other method is based on the use of block cipher algorithms (see Section 10.3.2). The block cipher derivation function specified in Section 10.3.2 uses a **BCC** function and a **Block_Encrypt** call that are discussed in Section 10.3.3.

The presence of these derivation functions in this Recommendation does not implicitly approve these functions for any other application.

10.3.1 Derivation Function Using a Hash Function (Hash_df)

This derivation function is used by the **Hash_DRBG** specified Section 10.1.1. 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 mechanism, and let *outlen* be its output length.

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

Hash_df (*input_string*, *no_of_bits_to_return***):**

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. temp = the Null string.
2. $len = \left\lceil \frac{no\_of\_bits\_to\_return}{outlen} \right\rceil$.
3. counter = 0x01.            Comment: An 8-bit binary value representing the
                              integer "1".
4. For i = 1 to len do
                              Comment : In step 4.1, no_of_bits_to_return is used
                              as a 32-bit string.
   4.1 temp = temp || Hash (counter || no_of_bits_to_return || input_string).
   4.2 counter = counter + 1.
5. requested_bits = leftmost (temp, no_of_bits_to_return).
6. Return (SUCCESS, requested_bits).
```

10.3.2 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. **BCC** and **Block_Encrypt** are discussed in Section 10.3.3. Let *outlen* be its output block length, which is a multiple of eight 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.

Block_Cipher_df (*input_string*, *no_of_bits_to_return***):**

1. *input_string*: The string to be operated on. This string **shall** be a multiple of eight 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
   and a Null string.
```

```
2. $L$ = len ($input\_string$)/8.        Comment: L is the bitstring representation 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 representation 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.
4. $S = L \| N \| input\_string \| 0x80$.
                                         Comment: Pad S with zeros, if necessary.
5. While (len ($S$) mod $outlen$) $\neq 0$, do
       $S = S \| 0x00$.
                                         Comment: Compute the starting value.
6. $temp$ = the $Null$ string.
7. $i = 0$.                              Comment: i shall be represented as a 32-bit integer,
                                         i.e., len (i) = 32.
8. $K$ = leftmost (0x00010203...1D1E1F, $keylen$).
9. While len ($temp$) < $keylen + outlen$, do
   9.1  $IV = i \| 0^{outlen - \mathbf{len}\,(i)}$.      Comment: The 32-bit integer representation of i is
                                         padded with zeros to outlen bits.
   9.2  $temp = temp \|$ BCC ($K$, ($IV \| S$)).
   9.3  $i = i + 1$.
                                         Comment: Compute the requested number of bits.
10. $K$ = leftmost ($temp$, $keylen$).
11. $X$ = select ($temp$, $keylen$+1, $keylen$+$outlen$).
12. $temp$ = the $Null$ string.
13. While len ($temp$) < $number\_of\_bits\_to\_return$, do
    13.1  $X$ = Block_Encrypt ($K$, $X$).
    13.2  $temp = temp \| X$.
14. $requested\_bits$ = leftmost ($temp$,  $number\_of\_bits\_to\_return$).
15. Return (SUCCESS, $requested\_bits$).
```

10.3.3 BCC and Block_Encrypt

Block_Encrypt is used for convenience in the specification of the **BCC** 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 **BCC** 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.

BCC (*Key*, *data*):

1. *Key*: The key to be used for the block cipher operation.
2. *data*: The data to be operated upon. Note that the length of *data* must be a multiple of *outlen*. This is guaranteed by **Block_Cipher_df** process steps 4 and 8.1 in Section 10.3.2.

Output:

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

BCC Process:

```
1.  $chaining\_value = 0^{outlen}$.        Comment: Set the first chaining value to outlen zeros.
2.  $n$ = len ($data$)/$outlen$.
3.  Starting with the leftmost bits of data, split data 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 ($output\_block$).
```

11 Assurance

A user of a DRBG employed for cryptographic purposes requires assurance that the generator actually produces (pseudo) 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 design of each DRBG mechanism in this Recommendation has received an evaluation of its security properties prior to its selection for inclusion in this Recommendation.

An implementer selects a DRBG mechanism (e.g., **HMAC_DRBG**), an appropriate cryptographic primitive (e.g., SHA-256 or SHA-512), the DRBG functions to be used (i.e., instantiate, generate and/or reseed), and will determine whether or not the DRBG will be distributed (see Section 8.5). Each choice of components effectively defines a different DRBG type. For example, an implementation of **HMAC_DRBG** using SHA-256 is considered to be a different DRBG than **HMAC_DRBG** using SHA-512.

An implementation **shall** be validated for conformance to this Recommendation by a NVLAP-accredited laboratory (see Section 11.2). Such validations provide a higher level of assurance that the DRBG mechanism is correctly implemented.

Health tests on the DRBG mechanism **shall** be implemented within a DRBG mechanism 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 mechanism **shall** be validated (see [FIPS 140]). The consuming application or cryptographic service that uses a DRBG mechanism **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 health testing) may be publicly known. Therefore, entropy input used for testing **shall not** be used for normal operational use.

11.1 Minimal Documentation Requirements

A set of documentation **shall** be developed that will provide assurance to users and validators that the DRBG mechanisms in this Recommendation have been implemented properly. Much of this documentation could be placed in a user 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 health testing.
- Document the type of DRBG mechanism (e.g., **CTR_DRBG**), and the cryptographic primitives used (e.g., AES-128 or SHA-256).
- Document the security strengths supported by the implementation.

- Document features supported by the implementation (e.g., prediction resistance, personalization string, additional input, etc.).
- If DRBG mechanism functions are distributed, specify the mechanisms that are used to protect the confidentiality and integrity of the internal state or parts of the internal state that are transferred between the distributed DRBG mechanism sub-boundaries (i.e., provide documentation about the secure channel).
- In the case of the **CTR_DRBG**, indicate whether a derivation function is provided. If a derivation function is not used, document that the implementation can only be used when full entropy input is available.
- Document any support functions other than health testing.
- If periodic testing is performed for the generate function, document the intervals and provide a justification for the selected intervals (see Section 11.3.3).
- Document whether the DRBG functions can be tested on demand.
- Document how the integrity of the health tests will be determined subsequent to implementation validation testing.

11.2 Implementation Validation Testing

A DRBG mechanism **shall** be tested for conformance to this Recommendation. A DRBG mechanism **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:

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

All DRBG functions included in an implementation **shall** be tested, including the health test functionality. The error handling of all implemented DRBG functions will be tested. See Section 11.4 for expected error handling behavior.

Note that when the uninstantiate function is tested, testing **shall** demonstrate that the internal state has been zeroized.

11.3 Health Testing

A DRBG implementation **shall** perform self-tests to obtain assurance that the DRBG continues to operate as designed and implemented (health testing). The testing function(s) within a DRBG mechanism boundary (or sub-boundary) **shall** test each DRBG mechanism function within that boundary (or sub-boundary), with the possible exception of the health test function itself. A DRBG implementation may optionally perform other self-tests for DRBG functionality in addition to the tests specified in this Recommendation.

The testing of the error handling capability is not required during the conduct of health tests. However, errors encountered during health testing **shall** be handled as discussed in Section 11.4.2.

All data output from the DRBG mechanism boundary (or sub-boundary) **shall** be inhibited while these tests are performed. The results from known-answer-tests (see Section 11.3.1) **shall not** be output as random bits during normal operation.

11.3.1 Known Answer Testing

Known-answer testing **shall** be conducted as specified below. A known-answer test involves operating the DRBG mechanism 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 mechanism **shall** enter an error state and output an error indicator (see Section 11.4).

Generalized known-answer testing is specified in Sections 11.3.2 through 11.3.5. With the possible exception of the health test function itself, testing **shall** be performed on all implemented DRBG mechanism functions within a DRBG boundary (if all functions are in the same device) or sub-boundary (if functions are distributed) (see Section 8.5). Documentation **shall** be provided that addresses the continued integrity of the health tests (see Section 11.1).

Known-answer tests **shall** be conducted on each DRBG function within a boundary or sub-boundary prior to the first use of that DRBG (e.g., during the power-on self-testing sequence).

11.3.2 Testing the Instantiate Function

Known-answer tests on the instantiate function **shall** use a security strength that will be available during normal operations. If prediction resistance has been implemented, the *prediction_resistance_flag* **shall** also be used. A representative fixed value and length of the *entropy_input*, *nonce* and *personalization_string* (if supported) **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).

If the values used during the test produce the expected results, then the instantiate function may be used during normal operation.

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

11.3.3 Testing the Generate Function

During generate-function testing, a representative fixed value and length for the *requested_number_of_bits* and *additional_input* (if supported) **shall** be used. If prediction resistance is supported, then the use of the *prediction_resistance_request* parameter **shall** be tested.

If the values used during the test produce the expected results, then the generate function may be used during normal operation.

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.

In addition to testing the generate function before first use (see Section 11.3.1), known-answer tests **should** be performed at reasonable intervals, as defined by the implementer.

11.3.4 Testing the Reseed Function

Known-answer testing of the reseed function **shall** use the *security_strength* in the internal state of the (testing) instantiation to be reseeded. A representative value of the *entropy_input* and *additional_input* (if supported) **shall** be used (see Sections 8.3 and 10). If prediction resistance for the reseed function is supported, then the use of the *prediction_resistance_request* parameter **shall** be tested.

If the values used during the test produce the expected results, then the reseed function may be used during normal operation.

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

11.3.5 Testing the Uninstantiate Function

Testing of the uninstantiate function is not required during health testing.

11.4 Error Handling

The expected errors are indicated for each DRBG mechanism function (see Sections 9.1 through 9.4) and for the derivation functions in Section 10.3. The error handling routines **should** indicate the type of error.

11.4.1 Errors Encountered During Normal Operation

Many errors that occur during normal operation may be caused by a consuming application's improper DRBG request or possibly the current unavailability of entropy; 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 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.

Catastrophic errors (i.e., errors indicated by the **CATASTROPHIC_ERROR_FLAG** in the pseudocode) detected during normal operation **shall** be treated in the same manner as an error detected during health testing (see Section 11.4.2).

11.4.2 Errors Encountered During Health Testing

Errors detected during health testing **shall** be perceived as catastrophic DRBG failures.

When a DRBG fails a health test or a catastrophic error is detected during normal operation, the DRBG **shall** enter an error state and output an error indicator. The DRBG **shall not** perform any instantiate, generate or reseed operations while in the error state, and pseudorandom bits **shall not** be output when an error state exists. When in an error state, user intervention (e.g., power cycling of the DRBG) **shall** be required to exit the error state, and the DRBG **shall** be re-instantiated before the DRBG can be used to produce pseudorandom bits. Examples of such errors include:

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

Appendix A: (Normative) Conversion and Auxiliary Routines

A.1 Bitstring to an Integer

Bitstring_to_integer ($b_1, b_2, \ldots, b_n$):

Input:

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

Output:

1. x The requested integer representation of the bitstring.

Process:

1. Let ($b_1, b_2, \ldots, b_n$) be the bits of a bitstring from leftmost to rightmost (i.e., most significant to least significant).
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 $x < 2^n$.

A.2 Integer to a Bitstring

Integer_to_bitstring (x**):**

Input:

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

Output:

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

Process:

1. Let ($b_1, b_2, \ldots, b_n$) 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 ($b_1, b_2, \ldots, b_n$).

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

A.3 Integer to a Byte String

Integer_to_byte_string (*x*):

Input:

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

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

Output:

1. A byte string *B* of length *n* bytes.

Process:

1. Let $B_1, B_2, \ldots, B_n$ be the bytes of *B* from leftmost to rightmost.
2. The bytes of *B* **shall** satisfy:

$$x = \sum_{i=1}^{n} 2^{8(n-i)} B_i.$$

3. Return (*B*).

A.4 Byte String to an Integer

Byte_string_to_integer (*B*):

Input:

1. A byte string *B* of length *n* bytes.

Output:

1. A non-negative integer *x*.

Process:

1. Let $B_1, B_2, \ldots, B_n$ be the bytes of *B* from leftmost to rightmost.
2. *x* is defined as follows:

$$x = \sum_{i=1}^{n} 2^{8(n-i)} B_i.$$

3. Return (*x*).

A.5 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 $(0 \leq s \leq r\text{-}1)$;

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 $0 \leq a_i \leq r\text{-}1$, 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 $0 \leq a_i \leq b\text{-}a$ (that is, when $r = b\text{-}a\text{+}1$).

A.5.1 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, $(b_0, b_1, \ldots, b_{m\text{-}1})$.
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.

A.5.2 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 $(a_0, a_1, \ldots , a_{t\text{-}1})$:

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

   let $(a_0, a_1, \ldots, a_{t\text{-}1})$ be the unique sequence of values satisfying $0 \leq a_i \leq r\ \text{-}1$ 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, … , $a_{t\text{-}1}$) 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 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 $0.5 < r^t/2^m \le 1$. 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 $t = 1$.

A.5.3 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, $(b_0, b_1, \ldots, b_{m+s\text{-}1})$.
2. Let $c = \sum_{i=0}^{m+s-1} 2^i b_i$.
3. Let a=c mod 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 ($0 \le w \le r\text{-}1$) 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.

Appendix B: (Informative) Example Pseudocode for Each DRBG Mechanism

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, arbitrary values have been selected that are consistent with the allowed values for each DRBG mechanism, as specified in the appropriate table in Section 10.

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

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.

B.1 Hash_DRBG Example

This example of **Hash_DRBG** uses the SHA-1 hash function, and prediction resistance is supported. Both a personalization string and additional input are supported. A 32-bit incrementing counter is used as the nonce for instantiation (*instantiation_nonce*); the nonce is initialized when the DRBG is instantiated (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 ten internal states are provided (i.e., ten 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 three input parameters, since the first two parameters (as specified in Section 9) have the same value.

The internal state contains values for *V*, *C*, *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 instantiated. 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.

B.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 has been designed to support 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 B.1.3.

Hash_DRBG_Instantiate_function:

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,
   prediction_resistance_request).
5. If (status ≠ "Success"), then Return (status, -1).
                                        Comment: Increment the nonce; actual coding must
                                        ensure that it wraps when the storage limit is
                                        reached.
6. instantiation_nonce = instantiation_nonce + 1.
                                        Comment: The instantiate algorithm is
                                        provided in steps 7 to 11.
7. seed_material = entropy_input || instantiation_nonce || personalization_string.
8. seed = Hash_df (seed_material, 440). Comment: Hash_df is defined in Section
                                        10.3.1.
9. V = seed.
10. C = Hash_df ((0x00 || V), 440).
11. reseed_counter = 1.
                                        Comment: Find an unused internal state.
12. (status, state_handle) = Find_state_space ( ).
13. If (status ≠ "Success"), then Return (status, -1).
14. Save the internal state.
    14.1 internal_state (state_handle).V = V.
    14.2 internal_state (state_handle).C = C.
    14.3 internal_state (state_handle).reseed_counter = reseed_counter.
    14.4 internal_state (state_handle). security_strength = security_strength.
    14.5 internal_state (state_handle).prediction_resistance_flag =
         prediction_resistance_flag.
15. Return ("Success", state_handle).
```

B.1.2 Reseeding a Hash_DRBG Instantiation

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

Hash_DRBG_Reseed_function:

Input: integer (*state_handle*, *prediction_resistance_request*), 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, 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.
         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,
   prediction_resistance_request).
5. If (status ≠ "Success"), then Return (status).
                              Comment: The reseed algorithm is provided in steps
                              6 to 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").
```

B.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_Generate_function:

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, 0,
    0, 0})), then Return ("State not available for the state_handle", Null).

2.  Get the internal state values.

    2.1 V = internal_state (state_handle).V.

    2.2 C = internal_state (state_handle).C.

    2.3 reseed_counter = internal_state (state_handle).reseed_counter.

    2.4 security_strength = internal_state (state_handle).security_strength.

    2.5 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 ((reseed_counter > 100 000) or (prediction_resistance_request = 1)), then

    6.1   status = Hash_DRBG_Reseed_ function (state_handle,
          prediction_resistance_request, additional_input).

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

    6.3   Get the new internal state values that have changed.

          7.3.1 V = internal_state (state_handle).V.

          7.3.2 C = internal_state (state_handle).C.

          7.3.3 reseed_counter = internal_state (state_handle).reseed_counter.

    6.4   additional_input = Null.

                                          Comment: Steps 7 to 15 provide the rest of the
                                          generate algorithm. Note that in this
                                          implementation, the Hashgen routine specified
                                          in Section 10.1.1.4 is provided inline as steps 8 to
                                          12.

7.  If (additional_input ≠ Null), then do
```

```
    7.1 $w$ = **Hash** (0x02 || $V$ || $additional\_input$).
    7.2 $V = (V + w) \bmod 2^{440}$.
8. $m = \left\lceil \frac{requested\_no\_of\_bits}{outlen} \right\rceil$.
9. $data = V$.
10. $W$ = the $Null$ string.
11. For $i$ = 1 to $m$
    11.1 $w$ = **Hash** ($data$).
    11.2 $W = W \| w$.
    11.3 $data = (data + 1) \bmod 2^{440}$.
12. $pseudorandom\_bits$ = **leftmost** ($W$, $requested\_no\_of\_bits$).
13. $H$ = **Hash** (0x03 || $V$).
14. $V = (V + H + C + reseed\_counter) \bmod 2^{440}$.
15. $reseed\_counter = reseed\_counter + 1$.
                                        Comments: Update the working_state.
16. Update the changed values in the state.
    16.1 internal_state (state_handle).V = V.
    16.2 internal_state (state_handle).reseed_counter = reseed_counter.
17. **Return** ("Success", pseudorandom_bits).
```

B.2 HMAC_DRBG Example

This example of **HMAC_DRBG** uses the SHA-256 hash function. Reseeding and prediction resistance are not supported. 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 **HMAC_DRBG_Update** function is specified in Section 10.1.2.2.

A personalization string is supported, but additional input is not. A total of three 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, and prediction resistance is not available.

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 bits may be instantiated. 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.

B.2.1 Instantiation of HMAC_DRBG

This implementation will return a text message and an invalid state handle (–1) when an error is encountered.

HMAC_DRBG_Instantiate_function:

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 if (requested_ security_strength ≤ 128), then security_strength = 128
     Else if (requested_ security_strength ≤ 192), then security_strength = 192
     Else security_strength = 256.
                                        Comment: Get the entropy_input and the
                                        nonce.
```

```
4. $min\_entropy = 1.5 \times security\_strength$.
5. ($status$, $entropy\_input$) = **Get_entropy_input** ($min\_entropy$, 1000).
6. If ($status \neq$ "Success"), then **Return** ($status$, −1).
                                   Comment: Invoke the instantiate algorithm. Note
                                   that the entropy_input contains the nonce.
7. ($V$, $Key$, $reseed\_counter$) = **HMAC_DRBG_Instantiate_algorithm** ($entropy\_input$,
   $personalization\_string$).
                                   Comment: Find an unused internal state.
8. ($status$, $state\_handle$) = **Find_state_space** ( ).
9. If ($status \neq$ "Success"), then **Return** ($status$, −1).
10. Save the initial state.
    10.1 $internal\_state$ ($state\_handle$).$V = V$.
    10.2 $internal\_state$ ($state\_handle$). $Key = Key$.
    10.3 $internal\_state$ ($state\_handle$). $reseed\_counter = reseed\_counter$.
    10.4 $internal\_state$ ($state\_handle$).$security\_strength = security\_strength$.
11. Return ("Success" and $state\_handle$).
```

HMAC_DRBG_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. ($Key$, $V$) = **HMAC_DRBG_Update** ($seed\_material$, $Key$, $V$).
5. $reseed\_counter = 1$.
6. **Return** ($V$, $Key$, $reseed\_counter$).
```

B.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_Generate_function:

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).
2. Get the internal state.
   2.1  V = internal_state (state_handle).V.
   2.2  Key = internal_state (state_handle).Key.
   2.3  security_strength = internal_state (state_handle).security_strength.
   2.4  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) =
   HMAC_DRBG_Generate_algorithm (V, Key, reseed_counter,
   requested_number_of_bits).
6. If (status = "Reseed required"), then Return ("DRBG can no longer be used. A new
   instantiation is required", Null).
7. Update the changed state values.
   7.1  internal_state (state_handle).V = V.
   7.2  internal_state (state_handle).Key = Key.
   7.3  internal_state (state_handle).reseed_counter = reseed_counter.
8. Return ("Success", pseudorandom_bits).
```

HMAC_DRBG_Generate_algorithm:

Input: bitstring (*V*, *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  temp = temp || V.
4. pseudorandom_bits = leftmost (temp, requested_no_of_bits).
5. (Key, V) = HMAC_DRBG_Update (Null, Key, V).
```

6. *reseed_counter* = *reseed_counter* + 1.
7. **Return** ("Success", *pseudorandom_bits*, *V*, *Key*, *reseed_counter*).

B.3 CTR_DRBG Example Using a Derivation Function

This example of **CTR_DRBG** uses AES-128 and uses the entire input block as the counter field. The reseed and prediction resistance capabilities are supported, and prediction resistance is obtained during every **Get_entropy_input** call and reseed request. Although the *prediction_resistance_request* parameter in the **Get_entropy_input** and reseed request could be omitted, in this case, they are shown in the pseudocode as a reminder that prediction_resistance will be performed. A block cipher derivation function using AES-128 is used, and a personalization string and additional input are supported. A total of five internal states are available. For this implementation, the functions and algorithms are written as separate routines. **AES_ECB_Encrypt** is the **Block_Encrypt** function (specified in Section 10.3.3) that 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 instantiated (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*, *reseed_counter*, and *security_strength*, where *V* and *Key* are bitstrings, and all other values are integers. Since prediction resistance is known to be supported, 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 bits 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. Input/output block length (*blocklen*) = 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.

B.3.1 The CTR_DRBG_Update Function

CTR_DRBG_Update:

Input: bitstring (*provided_data*, *Key*, *V*).

Output: bitstring (*Key*, *V*).

Process:

```
1. $temp = Null$.
2. While (len ($temp$) < 256) do
   2.1 $V = (V + 1) \bmod 2^{128}$.
   2.2 $output\_block$ = AES_ECB_Encrypt ($Key$, $V$).
   2.3 $temp = temp \| output\_block$.
3. $temp$ = leftmost ($temp$, 256).
4 $temp = temp \oplus provided\_data$.
5. $Key$ = leftmost ($temp$, 128).
6. $V$ = rightmost ($temp$, 128).
7. Return ($Key$, $V$).
```

B.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 10.3.2, 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 known to be supported.

CTR_DRBG_Instantiate_function:

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 ($requested\_instantiation\_security\_strength \le 112$), 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, prediction_resistance_request).
    5.  If (status ≠ "Success"), then Return (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, reseed_counter) = CTR_DRBG_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 (status, −1).
    10. Save the internal state.
        10.1  internal_state_ (state_handle).V = V.
        10.2  internal_state_ (state_handle).Key = Key.
        10.3  internal_state_ (state_handle).reseed_counter = reseed_counter.
        10.4  internal_state_ (state_handle).security_strength = security_strength.
    11. Return ("Success", state_handle).
```

CTR_DRBG_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) = CTR_DRBG_Update (seed_material, Key, V).
    6.  reseed_counter = 1.
    7.  Return (V, Key, reseed_counter).
```

B.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.

CTR_DRBG_Reseed_function:

Input: integer (*state_handle*), integer *prediction_resistance_request*, 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, 0, 0}), then Return ("State not available for the indicated state_handle").
2. Get the internal state values.
   2.1 V = internal_state (state_handle).V.
   2.2 Key = internal_state (state_handle).Key.
   2.3 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, prediction_resistance_request).
6. If (status $\neq$ "Success"), then Return (status).
                                        Comment: Invoke the reseed algorithm.
7. (V, Key, reseed_counter) = CTR_DRBG_Reseed_algorithm (V, Key,
   reseed_counter, entropy_input, additional_input).
8. Save the internal state.
   8.1 internal_state (state_handle). V = V.
   8.2 internal_state (state_handle). Key = Key.
   8.3 internal_state (state_handle). reseed_counter = reseed_counter.
   8.4 internal_state (state_handle). security_strength = security_strength.
9. Return ("Success").
```

CTR_DRBG_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) = CTR_DRBG_Update (seed_material, Key, V).
4. reseed_counter = 1.
5. Return (V, Key, reseed_counter).
```

B.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_Generate_function:

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, 0, 0}), then Return ("State not available for the indicated state_handle", Null).
2. Get the internal state.
   2.1  V = internal_state (state_handle).V.
   2.2  Key = internal_state (state_handle).Key.
   2.3  security_strength = internal_state (state_handle).security_strength.
   2.4  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 = CTR_DRBG_Reseed_function (state_handle,
        prediction_resistance_request, additional_input).
   7.2  If (status ≠ "Success"), then Return (status, Null).
   7.3  Get the new working state values; the administrative information was not
        affected.
        7.3.1  V = internal_state (state_handle).V.
        7.3.2  Key = internal_state (state_handle).Key.
        7.3.3  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, reseed_counter) =
   CTR_DRBG_Generate_algorithm (V, Key, 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. Update the internal state.
    10.1  internal_state (state_handle).V = V.
    10.2  internal_state (state_handle).Key = Key.
    10.3  internal_state (state_handle).reseed_counter = reseed_counter.
    10.4  internal_state (state_handle).security_strength = security_strength.
11. Return ("Success", pseudorandom_bits).
```

CTR_DRBG_Generate_algorithm:

Input: bitstring (*V*, *Key*), integer (*reseed_counter*, *requested_number_of_bits*) bitstring *additional_input*.

Output: string *status*, bitstring (*returned_bits*, *V*, *Key*), integer *reseed_counter*.

Process:

```
1. If (reseed_counter > 100 000), then Return ("Reseed required", Null, V, Key,
   reseed_counter).
2. If (additional_input ≠ Null), then
        2.1  additional_input = Block_Cipher_df (additional_input, 256).
        2.2  (Key, V) = CTR_DRBG_Update (additional_input, Key, V).
   Else additional_input = $0^{256}$.
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  temp = temp || output_block.
5. returned_bits = leftmost (temp, requested_number_of_bits)
```

6. (*Key*, *V*) = **CTR_DRBG_Update** (*additional_input*, *Key*, *V*)
7. *reseed_counter* = *reseed_counter* + 1.
8. **Return** (“Success”, *returned_bits*, *V*, *Key*, *reseed_counter*).

B.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 B.3, the **CTR_DRBG** uses AES-128. The reseed and prediction resistance capabilities are available. Both a personalization string and additional input are supported. A total of five internal states are available. For this implementation, the functions and algorithms are written as separate routines. **AES_ECB_Encrypt** is the **Block_Encrypt** function (specified in Section 10.3.3) that uses AES-128 in the ECB mode.

The internal state contains the values for *V*, *Key*, *reseed_counter*, and *security_strength*, where *V* and *Key* are strings, and all other values are integers. Since prediction resistance is known to be supported, 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 bits may be supported. The definitions are the same as those provided in Appendix B.3, except that to be compliant with Table 3, the maximum size of the *personalization_string* is 256 bits. In addition, the maximum size of any *additional_input* is 256 bits (i.e., **len** (*additional_input* ≤ *seedlen*)).

B.4.1 The CTR_DRBG_Update Function

The update function is the same as that provided in Appendix B.3.1.

B.4.2 Instantiation of CTR_DRBG Without a Derivation Function

The instantiate function (**CTR_DRBG_Instantiate_function**) is the same as that provided in Appendix B.3.2, except for the following:

- Step 2 is replaced by:

  If (**len** (*personalization_string*) > 256), then **Return** (“*Personalization_string* too long”, −1).

- Step 6 is replaced by :

  instantiation_nonce = *Null*.

The instantiate algorithm (**CTR_DRBG_Instantiate_algorithm**) is the same as that provided in Appendix B.3.2, except that steps 1 and 2 are replaced by:

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

If (*temp* < 256), then *personalization_string* = *personalization_string* || $0^{256-temp}$.

seed_material = *entropy_input* ⊕ *personalization_string*.

B.4.3 Reseeding a CTR_DRBG Instantiation Without a Derivation Function

The reseed function (**CTR_DRBG_Reseed_function**) is the same as that provided in Appendix B.3.3, except that step 3 is replaced by:

If (**len** (*additional_input*) > 256), then **Return** ("*additional_input* too long").

The reseed algorithm (**CTR_DRBG_Reseed_algorithm**) is the same as that provided in Appendix B.3.3, except that steps 1 and 2 are replaced by:

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

If (*temp* < 256), then *additional_input* = *additional_input* || $0^{256\text{-}temp}$.

seed_material = *entropy_input* ⊕ *additional_input*.

B.4.4 Generating Pseudorandom Bits Using CTR_DRBG

The generate function (**CTR_DRBG_Generate_function**) is the same as that provided in Appendix B.3.4, except that step 5 is replaced by:

If (**len** (*additional_input*) > 256), then **Return** ("*additional_input* too long", *Null*).

The generate algorithm (**CTR_DRBG_Generate_algorithm**) is the same as that provided in Appendix B.3.4, except that step 2.1 is replaced by:

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

If (*temp* < 256), then *additional_input* = *additional_input* || $0^{256\text{-}temp}$.

Appendix C: (Informative) DRBG Mechanism Selection

Almost no application or system designer starts with the primary purpose of generating good random bits. Instead, the designer typically starts with a goal that he wishes to accomplish, then decides on cryptographic mechanisms, such as digital signatures or block ciphers that can help him achieve that goal. Typically, as the requirements of those cryptographic mechanisms are better understood, he learns that random bits will need to be generated, and that this must be done with great care so that the cryptographic mechanisms will not be weakened. At this point, there are three things that may guide the designer's choice of a DRBG mechanism:

a. He may already have decided to include a set of cryptographic primitives as part of his implementation. By choosing a DRBG mechanism based on one of these primitives, he can minimize the cost of adding that DRBG mechanism. 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 mechanism (e.g., **Hash_DRBG** or **HMAC_DRBG**) is a natural choice.

b. He may already have decided to trust a block cipher, hash function, or keyed hash function to have certain properties. By choosing a DRBG mechanism based on similar properties, he can minimize the number of algorithms he has to trust.

   For example, an AES-based DRBG mechanism (i.e., **CTR_DRBG** using AES) might be a good choice when a module also provides encryption with AES. Since the security of the module 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).

   For example, a module with support for both hash functions and block ciphers might use the **CTR_DRBG** if the ability to parallelize the generation of random bits is needed.

The DRBG mechanisms specified in this Recommendation have different performance characteristics, implementation issues, and security assumptions.

C.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 [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 *outlen*-bit block of requested output, where *outlen* 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. The **Generate function** is parallelizable, since it uses the counter mode. Within a generate request, each *outlen*-bit block of output requires one hash function computation and several addition operations; 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 10.3.1) 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.

C.2 HMAC_DRBG

HMAC_DRBG is built around the use of an **approved** hash function using the HMAC construction. To generate pseudorandom bits from a secret key (*Key*) and a starting value *V*, the **HMAC_DRBG** computes

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

At the end of a generation request, the **HMAC_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** hash function used in the HMAC construction is a pseudorandom function family. Informally, this means that when an attacker does not 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 the pseudorandomness of HMAC is a widely used assumption in designs, 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.

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).

Algorithm Choices. The choice of hash functions that may be used by **HMAC_DRBG** is discussed in Section 10.1.

C.3 CTR_DRBG

CTR_DRBG is based on using an **approved** block cipher algorithm in counter mode (see [SP 800-38A]). At the present time, only three-key TDEA and AES are **approved** for use by the Federal government for use in this DRBG mechanism. 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 parallelizable. At the end of each generate request, work equivalent to two, three or four 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, as 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. As 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.

The output constraints are necessary to avoid a distinguishing attack on the **CTR_DRBG**, described in [Campagna], in which the fact that a single generate call can never produce a duplicate block from the block cipher is used to build a distinguisher for the DRBG's outputs. These output constraints apply to the use of **CTR_DRBG** for any single purpose, regardless of how many times the DRBG is reseeded. However, the distinguishing attack is theoretical − it poses no practical threat to any real-world application of the DRBG.

The distinguishing attack is conceptually quite simple. For concreteness, consider the case of TDEA **CTR_DRBG**. The DRBG generates a maximum of 128 64-bit blocks per generate request, thus providing 2^{13} bits per request. An ideal random source would have a very small probability (about 2^{-51}) of producing a pair of identical 64-bit blocks within that generate request output; each generate request from the **CTR_DRBG** is generated by running the block cipher in counter mode, so there can never be a duplicate block produced within a generate request output. (The block cipher is rekeyed between generate requests, so duplicate blocks can appear in different generate request outputs.) TDEA **CTR_DRBG** permits the use of up to 2^{32} generate requests. An ideal random source, providing 2^{32} sequences of 128 64-bit blocks, would have a probability of about 2^{-19} of having a duplicate block in one of those sequences of 128 64-bit blocks; the **CTR_DRBG** will never have such a duplicate block. This provides a distinguisher − an attacker, given a sequence of $2^{13}\ 2^{32} = 2^{45}$ bits from an ideal random source, has about a 2^{-19} probability of seeing an event happen that could never happen from TDEA **CTR_DRBG**.

Consider some application in which a DRBG's outputs must not be distinguishable by an attacker, and assume that an attacker who sees 2^{64} bits of output from the TDEA **CTR_DRBG** across at least one reseed, and wants to decide whether these bits came from the **CTR_DRBG** or from an ideal random source. The best case for the attacker is that each generate request used the maximum allowed value of 2^{13} bits of output = 128 64-bit blocks of output. In this case, the TDEA **CTR_DRBG** received 2^{45} generate requests. An ideal random sequence has a probability of about 2^{-6} of having a duplicate block in one of the generate outputs; the **CTR_DRBG** outputs will never have one. An attacker looking at the sequence will not be able to determine that it came from the **CTR_DRBG**, though he would have a pretty large advantage in a distinguishing game.

The case for AES **CTR_DRBG** is similar: each generate request may produce no more than 2^{19} bits, which means 2^{12} 128-bit blocks. In an ideal random sequence of 2^{12} 128-bit blocks, the probability that any two blocks will be the same is approximately 2^{-105}; AES **CTR_DRBG** will never provide a generate output with duplicate blocks. AES **CTR_DRBG** permits up to 2^{48} generate requests, so an attacker seeing the maximum length of output permitted (2^{67} bits) from either an AES **CTR_DRBG** instance or an ideal random sequence will have a 2^{-57} probability of being able to distinguish the two.

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 mechanism, but at a cost in performance because of the use of the derivation function (as opposed to not using the derivation function; see below). Such an implementation may be seeded by any **approved** randomness source 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 randomness source that provides full entropy (e.g., an **approved** entropy source that has full entropy output or an **approved** NRBG).

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).

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.

C.4 Summary for DRBG Selection

Table C-1 provides a summary of the costs and constraints of the DRBG mechanisms in this Recommendation.

Table C-1: DRBG Mechanism Summary

	Dominating Cost/Block	**Constraints (max.)**
Hash_DRBG	2 hash function calls	2^{48} calls of 2^{19} bits
HMAC_DRBG	4 hash function calls	2^{48} calls of 2^{19} bits
CTR_DRBG (TDEA)	1 TDEA encrypt	2^{32} calls of 2^{13} bits
CTR_DRBG (AES)	1 AES encrypt	2^{48} calls of 2^{19} bits

Appendix D : (Informative) References

[FIPS 140] Federal Information Processing Standard (FIPS) 140-2, *Security Requirements for Cryptographic Modules*, May 25, 2001 (including Change Notices as of December 3, 2002). http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf [accessed 6/9/15].

[FIPS 180] Federal Information Processing Standard (FIPS) 180-4, *Secure Hash Standard* (*SHS*), March 2012. http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf [accessed 6/9/15].

[FIPS 197] Federal Information Processing Standard (FIPS) 197, *Advanced Encryption Standard* (*AES*), November 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf [accessed 6/9/15].

[FIPS 198] Federal Information Processing Standard (FIPS) 198-1, *The Keyed-Hash Message Authentication Code* (*HMAC*), July 2008. http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf [accessed 6/9/15].

[SP 800-38A] National Institute of Standards and Technology Special Publication (SP) 800-38A, *Recommendation for Block Cipher Modes of Operation: Methods and Techniques*, December 2001. http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf [accessed 6/9/15].

[SP 800-38D] National Institute of Standards and Technology Special Publication (SP) 800-38D, *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*, November 2007. http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf [accessed 6/9/15].

[SP 800-57] NIST Special Publication (SP) 800-57 Part 1 Revision 3, *Recommendation for Key Management—Part 1*: General, July 2012. http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf [accessed 6/9/15].

[SP 800-67] NIST Special Publication (SP) 800-67 Revision 1, *Recommendation for the Triple Data Encryption Algorithm* (*TDEA*) *Block Cipher*, January 2012. http://csrc.nist.gov/publications/nistpubs/800-67-Rev1/SP-800-67-Rev1.pdf [accessed 6/9/15].

[SP 800-90B] NIST Special Publication (SP) 800-90B (Draft), *Recommendation for the Entropy Sources Used for Random Bit Generation*, August 2012 [re-released September 2013]. http://csrc.nist.gov/publications/drafts/800-90/draft-sp800-90b.pdf [accessed 6/9/15].

[SP 800-90C] NIST Special Publication (SP) 800-90C (Draft), *Recommendation for Random Bit Generator (RBG) Constructions*, August 2012 [re-released September 2013].

http://csrc.nist.gov/publications/drafts/800-90/draft-sp800-90c.pdf [accessed 6/9/15].

[SP 800-107] NIST Special Publication (SP) 800-107 Revision 1, *Recommendation for Applications Using Approved Hash Algorithms*, August 2012. http://csrc.nist.gov/publications/nistpubs/800-107-rev1/sp800-107-rev1.pdf [accessed 6/9/15].

[Campagna] M. J. Campagna, *Security Bounds for the NIST Codebook-based Deterministic Random Bit Generator*, Report 2006/379, Cryptology ePrint Archive, November 2006. http://eprint.iacr.org/2006/379 [accessed 6/9/15].

Appendix E : (Informative) Revisions

The original version of this Recommendation was completed in June, 2006. In **March 2007**, the following changes were made (note that the changes are indicated in italics):

1. Section 8.3, item 1.a originally stated the following:

   "One or more values that are derived from the seed and become part of the internal state; these values must usually remain secret"

   The item now reads:

   "One or more values that are derived from the seed and become part of the internal state; these values ***should*** remain secret".

2. In Section 8.4, the third sentence originally stated:

   "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."

   The sentence now reads:

   "Any security strength may be requested *(up to a maximum of 256 bits)*, but the DRBG will only be instantiated to one of the four security strengths above, depending on the DRBG implementation."

3. In Section 8.7.1, the list of examples of information that could appear in a personalization string included private keys, PINs and passwords. These items were removed from the list, and seedfiles were added.

4. In Section 10.3.1.4, a step was inserted that will provide backtracking resistance (step 14 of the pseudocode). The same change was made to the example in Appendix B.5.3 (step 19.1). In addition, the two occurrences of *block_counter* (in input 1 and processing step 1) were corrected to be *reseed_counter*.

This Recommendation was developed in concert with American National Standard (ANS) X9.82, a multi-part standard on random number generation. Many of the DRBGs in this Recommendation and the requirements for using and validating them are also provided in ANS X9.82, Part 3. Other parts of that Standard discuss entropy sources and RBG construction. During the development of the latter two documents, the need for additional requirements and capabilities for DRBGs were identified. As a result, the following changes were made to this Recommendation in **August 2008**:

1. Definitions have been added in Section 4 for the following: **approved** entropy source, DRBG mechanism, fresh entropy, ideal random bitstring, ideal random sequence and secure channel. The following definitions have been modified: backtracking resistance, deterministic random bit generator (DRBG), entropy, entropy input, entropy source, full entropy, min-entropy, prediction resistance, reseed, security strength, seed period and source of entropy input.

2. In Section 6, a link was provided to examples for the DRBGs specified in this Recommendation.

3. In Section 7.2, paragraph 3. 2nd sentence: The "**should**" has been changed to "**shall**", so that the sentence now reads:

   The personalization string **shall** be unique for all instantiations of the same DRBG mechanism type (e.g., **HMAC_DRBG**).

4. In Section 8.2, paragraph 2, additional text was added to the first sentence, which now reads:

   A DRBG is instantiated using a seed and may be reseeded*; when reseeded, the seed **shall** be different than the seed used for instantiation*.

5. In Section 8.5, Figure 4 has been updated, and the last paragraph has been revised to discuss the use of a secure channel.

6. In Sections 8.6.5 and 8.6.9, statements were inserted that prohibit a DRBG instantiation from reseeding itself.

7. References to "entropy input" have been removed from Section 8.6.9.

8. Section 8.8: An example was added to further clarify the meaning of prediction resistance.

9. In Section 9, a *prediction_resistance_request* parameter has been added to the **Get_entropy_input** call, along with a description of its purpose to the text underneath the call.

10. In Section 9, a footnote was inserted to explain why a *prediction_resistance_request* parameter may be useful in the **Get_entropy_input** call.

11. In Section 9.1, the following changes were made:

    - The following sentence has been added to the description of the *prediction_resistance_flag*:

      In addition, step 6 can be modified to not perform a check for the *prediction_resistance_flag* when the flag is not used in an implementation ; in this case, the **Get_entropy_input** call need not include the *prediction_resistance_request* parameter.

    - The following requirement has been added to the **Required information not provided by the consuming application during instantiation:**

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

    - A *prediction_resistance_request* parameter has been added to the **Get_entropy_input** call of step 6 of the **Instantiate Process**.

    - Step 5 was originally intended for implementations of the **Dual_EC_DRBG** to select an appropriate curve. This function is now performed by the **Dual_EC_DRBG**'s Instantiate_algorithm. Changes were made to provide the security strength to the Instantiate_algorithm. The Instantiate_algorithm for each DRBG was changed to allow the input of the security strength.

12. In Section 9.2, the following changes have been made:

- A *prediction_resistance_request* parameter has been added to the **Reseed_function** call.
- A description of the parameter has been added below the function call.
- A step was inserted that checked a request for prediction resistance (via the *prediction_resistance_request* parameter) against the state of the *prediction_resistance_flag* that may have been set during instantiation.
- A *prediction_resistance_request* parameter has been added to the **Get_entropy_input** call of (newly numbered) step 4 of the **Reseed Process**.
- In the description of the *entropy_input* parameter, a restriction was added that the *entropy_input* is not to be provided by the instantiation being reseeded.
- A footnote was inserted to explain why the prediction_resistance_request parameter might be useful.

13. In Section 9.3.1, the following changes were made:
    - Text has been added to item 4 to refer to the **Reseed_function**.
    - A *prediction_resistance_request* parameter has been added to the **Get_entropy_input** call of step 7.1 of the **Generate Process**.
    - A substep was inserted in step 9 of the **Generate Process** to check the *prediction_resistance request* against the state of the *prediction_resistance_flag*.
14. In Section 9.3.2, step e, a phrase addressing the presence of the *prediction_resistance_request* indicator was inserted.
15. In Sections 10.1 and 10.3.1, the new hash functions approved in FIPS 180-4 have been added.
16. In Sections 10.1.2 (**HMAC_DRBG**) and 10.2.1 (**CTR_DRBG**), the update functions have been renamed to reflect the DRBG with which they are associated (i.e., renamed to **HMAC_DRBG_Update** and **CTR_DRBG_Update**).
17. In Section 10.1.2.1, the last paragraph has been revised to indicate that only the Key is considered to be a critical value.
18. In Sections 10.1.2.3, 10.2.1.3.1, 10.2.1.3.2 and 10.3.1.2, the description of the *personalization_string* has been revised to indicate that the length the *personalization_string* may be zero.
19. In Section 10.2.1.5, the following statement has been added to the first paragraph:

    If the derivation function is not used, then the maximum allowed length of *additional_input = seedlen*.
20. In Section 10.3.1.2, the specification was changed to select an elliptic curve and return the parameters of that curve to the **Instantiate_function** that called the routine.
21. In the first paragraph of Appendix A.1, a statement has been added that if alternative points are desired, they **shall** be generated as specified in Appendix A.2.

22. The original Appendices C and D on entropy sources and RBG constructions, respectively, have been removed and the topics will be discussed in SP 800-90B and C
23. In Appendix C.2 (originally Appendix E.2), a paragraph has been inserted after the table of E values that discusses the analysis associated with the table values.
24. The additional uses of the *prediction_resistance_request* parameter (as specified in Section 9) have been added to the following appendices:
    - D.1.1, step 4;
    - D.1.2, Input and step 4;
    - D.1.3, step 7.1;
    - D.3.2, step 4;
    - D.3.3, Input and step 4; and
    - D.3.4, step 7.1.
25. The name of the update call has been changed in the following appendices:
    - D.2.1, step 4;
    - D.2.2, step 5;
    - D.3.1, title; and
    - D.4.1, title.
26. In Appendix D.3 (originally Appendix F.3), the first paragraph, which discusses the example, has been modified to discuss the *prediction_resistance_request* parameter in the **Get_entropy_input** call.
27. In Appendix D.5 (originally Appendix F.5), the description of the example in paragraph 2 has been changed so that the example does not include prediction resistance, and the definition for the *reseed_interval* has been removed from the list. The **Dual_EC_Instantiate_function** has been modified to reflect the changes made to the **Instantiate_function** and **Instantiate_algorithm** (see the last bullet of modification 8 above). In addition, the pseudocode for the **Reseed_function** has been removed, and steps in F.5.1 and F.5.2 that dealt with reseeding have been removed.

In **June 2015**, the following substantive changes were made in Revision 1 of [SP 800-90A]:

1. The following definitions were modified to be consistent with definitions in other parts of this Recommendation: backtracking resistance, entropy source, non-deterministic random bit generator, prediction resistance, and source of entropy input. The following definitions have been removed: public key and public-key pair. A definition for "randomness source" has been added, and the definition of "source of entropy input" has been removed.
2. The term "source of entropy input" has been replaced by "randomness source" to avoid confusion with the term "entropy source input," which is used in SP 800-90C to mean input from an entropy source. A "randomness source" (formerly "source of entropy input") could be an entropy source, an NRBG or a DRBG.

3. Section 5: The ECDLP abbreviation and the floor, ceiling and gcd symbols were removed. Definitions of the leftmost, rightmost, min and select functions have been added, and have been used throughout the document.
4. Section 6: The reference to number-theoretic problems was removed, as well as the old Appendix A that provided security considerations for DRBGs based on elliptic curves, and the old Appendix F that listed **shall** statements.
5. Section 7: The first paragraph has been modified, and includes an additional **shall** statement. In Section 7.1, the first two sentences have been modified for clarity. In Section 7.2, the second paragraph and the first sentence of the third paragraph have been modified for clarity; the personalization string is now recommended, rather than required, to be unique. In Section 7.4, the second item has been modified for clarity, and the last paragraph has been removed, since it was not needed here.
6. Section 8: In Section 8.1, the second sentence has been modified for clarity. In Section 8.2, additional text has been added to the last sentence for clarity. In Section 8.3, item 1b, the reference to *blocks* was removed, since it pertained to the Dual_EC_DRBG. In Section 8.4, the third sentence is a general statement that replaces the last two sentences of that paragraph; the subject with more detail is now discussed below Table 1. In the paragraph under Figure 4, text has been inserted in the second sentence for clarity. The first sentence of the next paragraph has been modified for clarity, and an additional paragraph has been added to the section to mention the relationship between a DRBG sub-boundary and a cryptographic module boundary.
7. Section 8.5: A reference to the cryptographic boundary for FIPS 140 has been inserted in **bold** to draw the reader’s attention to the fact that it is different than the DRBG’s boundaries. In the paragraph under item 3, an example has been provided for clarity. In the following two paragraphs, a reference to SP 800-90C has been inserted to direct the reader to that document for further discussion on cryptographic module boundaries.
8. Section 8.6: In Section 8.6.2, a reference to fresh entropy has been inserted in the second sentence. In Section 8.6.3, text has been inserted at the end of the second sentence for clarity. In Section 8.6.4, a **shall** statement has been inserted at the end of the first sentence. Sections 8.6.5 and 8.6.7 were revised to clarify the source of the entropy input and nonce. In Section 8.6.6, text was inserted that states that entropy input is a critical security parameter for cryptographic module validation. Section 8.6.7 was modified to provide more information about suitable nonces and to state that the uniqueness of the nonce is applicable to the cryptographic module in which it is used, and to indicate that the nonce is a critical security parameter. In Section 8.6.8, text was added about enforcing the seedlife. In Section 8.6.9, ‘DRBG’ was changed to ‘DRBG instantiation’ for clarity.
9. Section 8.7: Sections 8.7.1 and 8.7.2 have been modified to clarify that the optional personalization string and additional input may be obtained from outside a cryptographic module, that the personalization string is not a critical security parameter, and that the additional input may be a critical security parameter if secret information is included.
10. Section 8.8: The last sentence of the second paragraph under the list has ‘direct or indirect’ inserted for clarity. A paragraph has been added to the end of the section to recommend reseeding whenever possible.

11. Section 9: A paragraph discussing the pseudocode used has been inserted at the beginning of the section, and modifications to the third and fourth paragraphs have been made for clarity; text has also been added to the next-to-last paragraph that discusses error codes more thoroughly. The last sentence in the third paragraph has been modified to only require that the entropy input and nonce be provided as discussed in Sections 8.6.5 and 8.6.7 and in SP 800-90C. A paragraph has been added to discuss checking the status code. In Section 9.2, clarifying information has been inserted about the *prediction_resistance_request* parameter. In Sections 9.1, 9.2 and 9.3, returns to the consuming application have been modified for those cases where other than SUCCESS is appropriate as a status to be returned from the function (e.g., parameter errors, entropy unavailability or entropy source failure); this change was made to better accommodate the various **Get_entropy_input** constructions specified in SP 800-90C. In Section 9.1 and 9.3.1, the item in the list referring to elliptic-curve parameters was removed, and the discussion of the *status* output has been modified for clarity.

12. Section 10: Section 10 now includes a link to the DRBG test vectors on the NIST web site.

    Sections 10.1, 10.1.1 and 10.1.2 now include short discussions about selecting hash functions to support the DRBG's intended security strength.

    The **Dual_EC_DRBG** has been removed, and section numbers adjusted accordingly. In Section 10.2.1, a paragraph under Table 3 has been added for explanatory purposes. In Section 10.2.1.3.2, the first paragraph has been modified for clarity. Section 10.2 has been modified to allow the counter field to be a subset of the input block and to allow either derivation function specified in the document; this is indicated in step 2.1 of Section 10.2.1.2 and step 4.1 of Sections 10.2.1.5.1 and 10.2.1.5.2 (note that this change continues to allow the use of the entire input block as the counter field, as was specified in the previous versions of this document); Table 3 has been modified to include restrictions on the length of the counter field and to indicate the restrictions on the number of bits that can be requested during a single request as a function of the counter-field length and the previous restriction on the number of bits that could be requested. The first paragraphs of Sections 10.3 and 10.3.2 have been modified slightly for clarity.

    Step 11 in Section 10.3.2 has been respecified using the (new) select function.

13. Section 11: The third paragraph has been added for clarity, and the last sentence of the next paragraph has been removed. In Section 11.1, the references to the **Dual_EC_DRBG** have been removed from the third and fifth bullet, and the wording of the next-to-last bullet has been modified to be conditional. In Section 11.2, additional text has been inserted to address validation testing. In Section 11.3, the health testing requirements have been modified.

14. The previous Appendix A was removed; this appendix contained application-specific constants for the **Dual_EC_DRBG**.

15. Appendix A now contains the conversion routines. Appendix A.5.4 (the old Appendix B.5.4), which contained the complex modular method for converting bits to numbers, has been removed because of an error in the specification.

16. Appendix B now contains the pseudocode examples previously provided in Appendix D, less examples for the **Dual_EC_DRBG**. In Appendix B.4, the discussion of the example has been changed slightly.
17. The previous Appendix C was removed; this appendix contained security considerations relating to the **Dual_EC_DRBG**.
18. The new Appendix C is the same as the previous Appendix E, minus the **Dual_EC_DRBG** discussion.

    Additional text has been inserted into the discussion of the **CTR_DRBG** in Appendix C.3 (the constraints subsection) that discusses the constraints provided in Table 3 of Section 10.2.1.
19. The referenced documents now in Appendix D have been updated, and a reference to [Campagna] has been added.
20. The previous Appendix F was removed; this appendix contained a list of **shall** statements that could not be validated by NIST’s validation program.