(Second Draft) NIST Special Publication 800-90C

Recommendation for Random Bit Generator (RBG) Constructions

Elaine Barker
John Kelsey

C O M P U T E R   S E C U R I T Y

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

(Second Draft) NIST Special Publication 800-90C

Recommendation for Random Bit Generator (RBG) Constructions

Elaine Barker
John Kelsey
Computer Security Division
Information Technology Laboratory

April 2016

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

This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Modernization Act (FISMA) of 2014, 44 U.S.C. § 3541 *et seq.*, Public Law (P.L.) 113-283. NIST is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130.

Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on Federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other Federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

National Institute of Standards and Technology Special Publication 800-90C
Natl. Inst. Stand. Technol. Spec. Publ. 800-90C, 88 pages (April 2016)
CODEN: NSPUE2

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies, may be used by federal agencies even before the completion of such companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely follow the development of these new publications by NIST.

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at http://csrc.nist.gov/publications.

Public comment period: ***April 13, 2016 through June 13, 2016***

All comments are subject to release under the Freedom of Information Act (FOIA).

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

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in Federal information systems. The Special Publication 800-series reports on ITL's research, guidelines, and outreach efforts in information system security, and its collaborative activities with industry, government, and academic organizations.

Abstract

This Recommendation specifies constructions for the implementation of random bit generators (RBGs). An RBG may be a deterministic random bit generator (DRBG) or a non-deterministic random bit generator (NRBG). The constructed RBGs consist of DRBG mechanisms, as specified in NIST Special Publication (SP) 800-90A, and entropy sources, as specified in SP 800-90B.

Keywords

Construction; deterministic random bit generator (DRBG); entropy; entropy source; non-deterministic random bit generator (NRBG); random number generator; randomness source.

Acknowledgements

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

Table of Contents

1 Scope

Cryptography and security applications make extensive use of random bits. However, the generation of random bits is problematic in many practical applications of cryptography. The purpose of this Recommendation is to specify **approved** random bit generators (RBGs). By matching the security requirements of the application using the random bits with the security claims of the RBG generating those bits, an application can safely use the random bits produced by an RBG conforming to this Recommendation.

NIST Special Publications (SPs) 800-90A and SP 800-90B have addressed the components of RBGs:

- SP 800-90A, *Random Number Generation Using Deterministic Random Bit Generator Mechanisms*, specifies several Deterministic Random Bit Generator (DRBG) mechanisms containing **approved** cryptographic algorithms.
- SP 800-90B, *Recommendation for the Entropy Sources Used for Ransom Bit Generation*, provides guidance for the development and validation of entropy sources – mechanisms that generate randomness from a physical phenomenon.

SP 800-90C specifies the construction of **approved** RBGs using the DRBG mechanisms and entropy sources from SP 800-90A and SP 800-90B, respectively. SP 800-90C is based on American National Standard (ANS) X9.82, Part 4, and specifies constructions for an RBG, as well as constructions for building components that are used within those RBG constructions.

Throughout this document (i.e., SP 800-90C), the term "this Recommendation" refers to the aggregate of SP 800-90A, SP 800-90B and SP 800-90C.

The information in SP 800-90C is intended to be combined with the information in SP 800-90A and SP 800-90B in order to:

- Construct an RBG with the required security properties, and
- Verify that an RBG has been constructed in compliance with this Recommendation.

The precise structure, design and development of an RBG are outside the scope of this Recommendation.

2 Terms and Definitions

Approved	FIPS-**approved**, NIST-Recommended and/or validated by the Cryptographic Algorithm Validation Program (CAVP) or Cryptographic Module Validation Program (CMVP).
Approved DRBG	A DRBG implementation that uses an **approved** DRBG mechanism, an **approved** entropy source, and a DRBG construction that has been validated as conforming to SP 800-90C.
Approved DRBG mechanism	A DRBG mechanism that has been validated as conforming to SP 800-90A.
Approved entropy source	An entropy source that has been validated as complying with SP 800-90B.
Approved NRBG	An NRBG that uses an **approved** DRBG mechanism, an **approved** entropy source, and an NRBG construction that has been validated as conforming to SP 800-90C.
Approved RBG	An **approved** DRBG or an **approved** NRBG.
Backtracking resistance	A property whereby an attacker with 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*.)
Big-endian format	The most significant bytes (the bytes containing the high order or leftmost bits) are stored in the lowest address, with the following bytes in sequentially higher addresses.
Bits of security	See Security strength.
Bitstring	An ordered sequence (string) of 0's and 1's.
Chain of RBGs (or DRBGs)	A succession of RBGs where the randomness source for one DRBG is another DRBG, NRBG or entropy source.

Conditioning function	An optional component that is used to process a bitstring containing entropy to reduce the bias and/or distribute the entropy across the output of the conditioning function.
Construction	A specific method of designing an RBG or some component of an RBG to accomplish a stated goal.
Consuming application	An application that uses the output from an **approved** random bit generator.
Derivation function	A function that is used to either derive internal state values, to distribute entropy throughout a bitstring or to compress the entropy in a bitstring into a shorter bitstring of a specified length.
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 Bit (or Number) Generator. (Contrast with a *Non-deterministic random bit generator* (*NRBG*)).
DRBG mechanism	The portion of an RBG that includes the functions necessary to instantiate and uninstantiate a DRBG, generate pseudorandom bits, test the health of the DRBG mechanism, and (optionally) reseed the DRBG. DRBG mechanisms are specified in SP 800-90A.
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	The combination of a noise source (e.g., thermal noise or hard drive seek times), health tests, and an optional conditioning component that produces the random bitstrings to be used by an RBG.
Equivalent process	A process that produces the same output as another process, given the same input as the other process.
External conditioning	The use of a conditioning function on the output of an entropy source prior to its use by other components of an RBG. Note

	that the entropy-source output may or may not have been conditioned within the entropy source. See *Internal conditioning*.
Fresh entropy	A bitstring output from a randomness source for which there is a negligible probability that it has been previously output by the source and a negligible probability that the bitstring has been previously used by the RBG.
Full-entropy output	Output that cannot be distinguished from a sequence of bits of the same length produced by an ideal random-number source with a probability substantially higher than 1/2. (See *Ideal random sequence*.)
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 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.
Independent entropy sources	Entropy sources that have no overlap of their security boundaries.
Independent randomness sources	The probability of correctly predicting the output of any given randomness source is unaffected by knowledge of the output of any or all other randomness sources.
Instantiate	The process of initializing a DRBG with sufficient entropy to generate pseudorandom bits at the desired security strength.
Internal conditioning	The use of a conditioning function to process the output of a noise source within an entropy source prior to providing entropy-source output.

Keying material	The data (e.g., keys, certificates, and initialization vectors) necessary to establish and maintain cryptographic keying relationships.
Known-answer test	A test that uses a fixed input/output pair to detect whether a component was implemented correctly or to detect whether it continues to operate correctly.
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 (in bits)	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 probabilities $p_1, ..., p_k$. Min-entropy is often used as a worst-case measure of the unpredictability of a random variable. (Also, see *Entropy*.)
Narrowest internal width	The maximum amount of information from the input that can affect the output. For example, if $f(x) = SHA\text{-}1(x) \| 01$, and x consists of a string of 1000 binary bits, then the narrowest internal width of $f(x)$ is 160 bits (the SHA-1 output length), and the output width of $f(x)$ is 162 bits (the 160 bits from the SHA-1 operation, concatenated by 01).
Nonce	A time-varying value that has at most a negligible chance of repeating.
Noise source	The component of an entropy source that contains the non-deterministic, entropy-producing activity.
Non-deterministic Random Bit Generator (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* (*DRBG*)).

Null string The empty bitstring.

Prediction resistance A property whereby 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.)

Random Bit Generator (RBG) A device or algorithm that is capable of producing a random sequence of (what are effectively indistinguishable from) statistically independent and unbiased bits. An RBG is classified as either a DRBG or an NRBG.

Randomness source A component of an RBG that outputs bitstrings that can be used as entropy input by a DRBG mechanism.

Reseed To acquire additional bits with sufficient entropy for the desired security strength.

Reseed interval The period of time between instantiating or reseeding a DRBG with one seed and reseeding that DRBG with another seed.

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, logical or procedural methods, or a combination thereof. Somestimes called a trusted channel.

Security boundary (of an entropy source) A conceptual boundary that is used to assess the amount of entropy provided by the values output from an entropy source. The entropy assessment is performed under the assumption that any observer (including any adversary) is outside of that boundary.

Security strength A number associated with the amount of work (that is, the number of basic 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, and 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.

Source RBG An RBG that is used directly as a randomness source.

Threat model A description of a set of security aspects that need to be considered; a threat model can be defined by listing a set of possible attacks, along with the probability of success and potential harm from each attack.

Uninstantiate The process of removing a DRBG from use by zeroizing the internal state of the DRBG.

3 Symbols and Abbreviated Terms

The following abbreviations are used in SP 800-90C.

Symbols and Abbreviations	Meaning
AES	Advanced Encryption Standard.
ANS	American National Standard.
CAVP	Cryptographic Algorithm Validation Program.
CTR_DRBG	A DRBG specified in SP 800-90A that is based on block cipher algorithms.
DRBG	Deterministic Random Bit Generator.
FIPS	Federal Information Processing Standard.
HMAC_DRBG	A DRBG specified in SP 800-90A that is based on HMAC.
NIST	National Institute of Standards and Technology.
NRBG	Non-deterministic Random Bit Generator.
RBG	Random Bit Generator.
RNG	Random Number Generator.
SP	Special Publication.
XOR-NRBG	NRBG construction that uses a bitwise exclsing-or operation.

The following symbols and function calls are used in SP 800-90C.

Symbol	Meaning
leftmost (*V*, *a*)	Selects the leftmost *a* bits of the bitstring *V*, i.e., the most significant *a* bits of *V*.
min(*a*, *b*)	The minimum of the two values *a* and *b*.
max(*a*, *b*)	The maximum of the two values *a* and *b*.
s	Security strength
$X \oplus Y$	Boolean bitwise exclusive-or (also bitwise addition modulo 2) of two bitstrings *X* and *Y* of the same length.
$X \| Y$	Concatenation of two bitstrings *X* and *Y*.
+	Addition over non-negative integers.
0^x	A string of *x* zero bits.
×	Multiplication over non-negative integers.

4 General Discussion

An RBG that conforms to this Recommendation produces random bits for a consuming application. The security of the RBG depends on:

- A deterministic process (the RBGs currently specified in SP 800-90C include DRBG mechanisms as discussed and specified in SP 800-90A) and
- A randomness source (e.g., an entropy source as specified in SP 800-90B or another RBG as specified in this document).

There are two classes of RBGs specified in SP 800-90C: Non-deterministic Random Bit Generators (NRBGs) and Deterministic Random Bit Generators (DRBGs). The choice of using an NRBG or DRBG may be based on the following:

- NRBGs provide full-entropy output. See Section 5.2 for a discussion of full entropy, and Sections 5.6 and 9 for discussions of NRBGs. The security strength that can be provided by any output of an NRBG is equal to the length of that output[1].
- DRBGs provide output that cannot be distinguished from an ideal random sequence without an infeasible amount of computational effort. When designed and used as specified in this Recommendation, DRBGs have a fixed (finite) security strength, which is a measure of the amount of work required to defeat the security of the DRBG. See Sections 5.5 and 8 for discussions of DRBGs.

  DRBGs are divided into two types: those that can provide prediction resistance, and those that cannot. See Section 5.4 for a discussion of prediction resistance.

4.1 RBG Security

Any failure of an RBG component could affect the security provided by the RBG. Any RBG designed to comply with this Recommendation will function at the designed security strength only if the following requirements are satisfied.

1. Entropy sources **shall** comply with SP 800-90B.
2. DRBG mechanisms **shall** comply with SP 800-90A.
3. Every DRBG **shall** be instantiated using an appropriate randomness source (see Section 6).
4. RBG boundaries **shall** include mechanisms that either detect or prevent access to RBG components from outside the boundary with respect to a specific threat model (see Section 5.1).
5. Bitstrings containing entropy **shall** only be used once.

[1] Note that the security strength of a string greater than 256 bits in length will provide a security strength greater than the highest security strength currently specified for Federal applcations (i.e., 256 bits).

4.2 Assumptions

The RBG constructions in SP 800-90C are based on the following assumptions:

1. Each output from an entropy source has a fixed length, *ES_outlen* (in bits).
2. Each output from an entropy source has a fixed amount of entropy, *ES_entropy*, that was assessed during entropy-source implementation validation.
3. Entropy-source output can be collected from a single entropy source to form a bitstring that is longer than a single output by concatenating the outputs. The entropy of the resultant bitstring is the sum of the entropy from each entropy-source output. For example, if three outputs from the same entropy source are concatenated, then the length of the bitstring is 3 × *ES_outlen* bits, and the entropy for that bitstring is 3 × *ES_entropy* bits.
4. Entropy-source output can be collected from multiple independent entropy sources. If the entropy sources are independent (i.e., their security boundaries do not overlap), then the outputs may be concatenated to form a single bitstring. The entropy in the resultant bitstring is the sum of the entropy from each entropy-source output that contributed entropy to the bitstring. For example, if the output from entropy sources A and B are concatenated, the length of the resulting bitstring is $ES\_outlen_A + ES\_outlen_B$, and the amount of entropy is $ES\_entropy_A + ES\_entropy_B$.
5. An entropy source is capable of providing a) an indication of success and the requested amount of entropy, or b) an indication of a failure (see Section 12.1.3 for a discussion of handling an entropy source failure).
6. The output of an entropy source (or the concatenated output of multiple entropy sources) can be externally conditioned to reduce residual bias or to condense the entropy into a shorter bitstring.
7. Under the right conditions, the output of an entropy source can be externally conditioned to provide full-entropy outputs. This requires several conditions to be met, including a requirement that the entropy-source output that is provided as input to the conditioning function have at least twice the amount of entropy as the number of bits that are produced as output from the conditioning function (see Section 5.3.5 for further discussion).
8. The DRBG mechanisms specified in SP 800-90A meet their explicit security claims (e.g., backtracking resistance, claimed security strength, etc.).

4.3 Constructions

SP 800-90C provides constructions for designing and implementing DRBGs and NRBGs from components specified in SP 800-90A and SP 800-90B. A construction is a method of designing an RBG or some component of an RBG to accomplish a specific goal. One or more of the constructions provided herein **shall** be used in the design of an RBG that conforms to this Recommendation. Each construction is intended to describe the behavior intended for the process; a developer may implement the construction as described or may implement an equivalent process. Two processes are equivalent if, when the same values are input to each process, the same output is produced.

Constructions are specified in SP 800-90A for the instantiation, generation of (pseudo) random output, reseeding and uninstantiation of a DRBG, and further details are discussed in Section 8. During instantiation, a DRBG is seeded with the amount of entropy needed to provide output at a given maximum security strength. Once instantiated, a DRBG can generate output at a security strength that does not exceed the DRBG's instantiated security strength. Reseeding is used to insert additional entropy into a DRBG. Uninstantiation is used to terminate a DRBG instantiation.

Two constructions for NRBGs are provided in Section 9:

- Section 9.3 specifies constructions for the XOR-NRBG, in which the output of an entropy source is exclusive-ORed with the output of a DRBG:
- Section 9.4 specifies constructions for the Oversampling-NRBG, which accesses an entropy source from a DRBG in a way that provides the full-entropy output required from an NRBG.

For each NRBG, constructions are provided to instantiate the NRBG (**NRBG_Instantiate**) and request NRBG output (**NRBG_Generate**).

Additional constructions are used by the DRBG or NRBG to acquire entropy input from a randomness source using a **Get_entropy_input** call. A randomness source can be either an entropy source or another RBG.

- Section 10.1 provides **Get_entropy_input** constructions to use a DRBG as randomness source; the consruction to be used depends on the security strength to be requested and whether prediction resistance is required.
- Section 10.2 provides a **Get_entropy_input** construction for using an NRBG as a randomness source.
- Section 10.3 provides several **Get_entropy input** constructions for accessing an entropy source as the randomness source. Also included are constructions for condensing entropy-source output when the output has sparse entropy. The output from an entropy source may also be conditioned prior to use by an RBG; constructions for vetted conditioning functions are provided in SP 800-90B.

A construction is also provided for obtaining full-entropy output from a DRBG when that DRBG can provide prediction resistance and an entropy source is available (see Section 10.4).

The output of RBGs may also be combined, as long as at least one RBG is compliant with SP 800-90. Section 11 provides constructions for instantiating, reseeding and generating output from multiple RBGs.

4.4 Document Organization

The remainder of SP 800-90C describes how to construct an RBG from the components described in SP 800-90A and SP 800-90B.

Section 5 provides RBG concepts, such as RBG boundaries, distributed RBGs, full entropy, live entropy sources, prediction resistance, and introductory discussions on DRBGs and NRBGs.

Section 6 provides an overview of the randomness sources to be used by a DRBG.

Section 7 describes the conceptual interface calls used in SP 800-90C.

Sections 8 and 9 provide guidance for constructing DRBGs and NRBGs, respectively.

Section 10 provides constructions for implementing a DRBG's **Get_entropy_input** call using DRBGs, NRBGs and entropy sources as randomness sources. Section 10 also discusses the use of **approved** functions for conditioning entropy-source output.

Section 11 provides guidance on combining RBGs.

Section 12 discusses testing, including both health testing and implementation-validation testing.

Appendix A contains examples of RBG configurations.

Appendix B contains a list of references.

Additional material is addressed in American National Standard (ANS) X9.82, Part 4, including expanded explanations and:

- A step-by step description for constructing an RBG,
- Obtaining entropy from entropy sources that are only available intermittently, and
- Security and implementation considerations.

5 Random Bit Generator Concepts

5.1 RBG Boundaries and Distributed RBGs

RBGs **shall** be implemented within FIPS 140-validated cryptographic modules (see Section 12). These cryptographic modules are defined with respect to cryptographic-module boundaries (see [FIPS 140]).

An RBG **shall** exist within a *conceptual* RBG security boundary that is defined with respect to one or more threat models, which include an assessment of the applicability of an attack and the potential harm caused by the attack. The RBG boundary **shall** be designed to assist in the mitigation of these threats, using either physical or logical mechanisms or both.

An RBG boundary **shall** contain all components required for the RBG. Data **shall** enter an RBG only via the RBG's public input interface(s) (if any) and **shall** exit only via its public output interface(s). The primary components of an RBG are a randomness source (e.g., an entropy source), a DRBG mechanism and health tests for the RBG. The boundaries of a DRBG mechanism are discussed in SP 800-90A. The security boundary for an entropy source is discussed in SP 800-90B. Both the entropy source and the DRBG mechanism contain their own health tests within their respective boundaries. Note that the RBG boundary consists of at least two conceptual sub-boundaries: a boundary for a DRBG mechanism, and a boundary for the source of randomness (e.g., an entropy source).

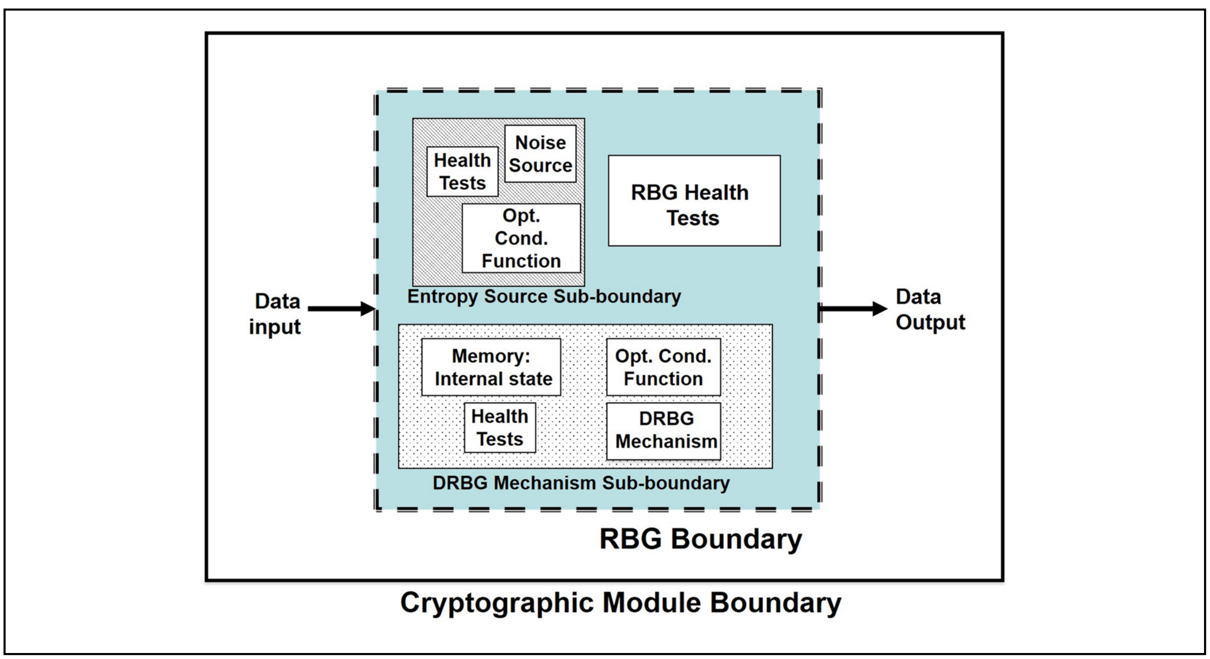

Figure 1: RBG within a Single Cryptographic Module

An RBG may be implemented within a single cryptographic module, as shown in Figure 1. In this case, the RBG boundary is either the same as the cryptographic module boundary or is completely contained within that boundary. Within the RBG boundary are an entropy source and a DRBG mechanism, each with its own (conceptual) sub-boundary. The entropy-source sub-boundary includes a noise source, health tests and optionally, a conditioning function. The sub-boundary for the DRBG mechanism contains the chosen DRBG mechanism, an optional conditioning function, memory for the internal state and health tests. The RBG boundary also contains its own health tests.

Alternatively, an RBG may be distributed among multiple cryptographic modules; an example is shown in Figure 2. In this case, each cryptographic module **shall** have an RBG sub-boundary that contains the RBG component(s) within that module. The RBG component(s) within each sub-boundary are protected by the cryptographic module boundary that contains those RBG components. Test functions **shall** be provided within each sub-boundary to test the health of the RBG component(s) within that sub-boundary. Communications between the sub-boundaries (i.e., between the cryptographic modules) **shall** use reliable secure channels that provide confidentiality, integrity and replay protection of the data transferred between the sub-boundaries, as well as mutual authentication between the entities or components. The boundary for a distributed RBG encapsulates the contents of the cryptographic module boundaries and RBG sub-boundaries, as well as the secure channels. The security provided by a distributed RBG is no more than the security provided by the secure channel(s) and the cryptographic modules.

In the example in Figure 2, the entropy source is contained within a single RBG sub-boundary within one cryptographic module (indicated by the dotted-line box), while the DRBG mechanism is distributed across other sub-boundaries within other cryptographic modules (see SP 800-90A for further discussion of a distributed DRBG mechanism boundary). Secure channels are provided between the cryptographic modules to transport requests and responses between the RBG sub-boundaries.

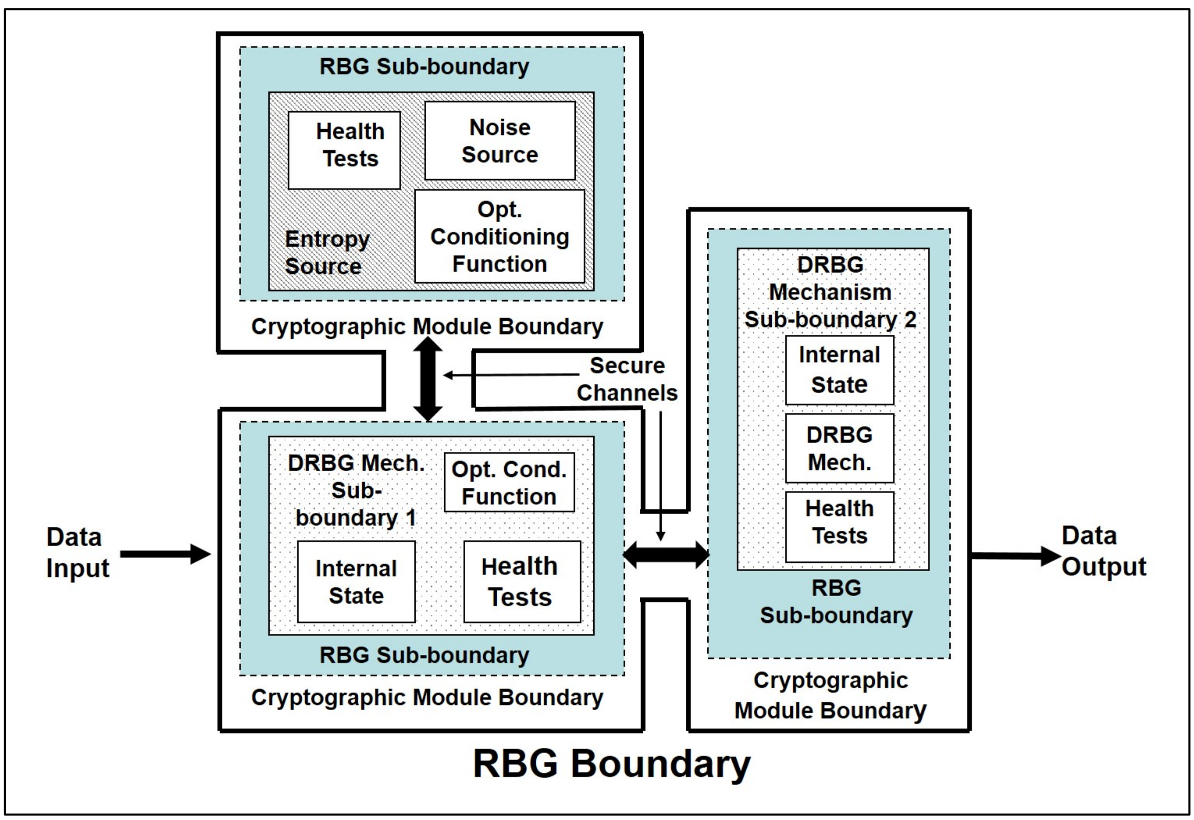

Figure 2: Distributed RBG

When an RBG uses cryptographic primitives (e.g., an **approved** hash function), other applications within the cryptographic module containing that primitive may use the same implementation of the primitive, as long as the RBG's output and internal state are not modified or revealed by this use.

5.2 Full Entropy

Each bit of a bitstring with full entropy has a uniform distribution and is independent of every other bit of that bitstring. Simplistically, this means that a bitstring has full entropy if every bit of the bitstring has one bit of entropy; the amount of entropy in the bitstring is equal to its length.

For the purposes of this Recommendation, an *n*-bit string is said to have full entropy if the string is the result of an **approved** process whereby the entropy in the input to that process has at least 2*n* bits of entropy (see [ILL89] and Section 4.2). Full-entropy output could be provided by an entropy source for use in an RBG (see SP 800-90B), by the output of an external conditioning function using the output of an entropy source (see Section 10.3), by a properly constructed DRBG (see Sections 10.1.2 and 10.4) or by an NRBG (see Sections 5.6 and 9).

5.3 Entropy Sources

5.3.1 Approved Entropy Sources

SP 800-90B discusses entropy sources. An entropy source is considered **approved** if it has been successfully validated as conforming to SP 800-90B.

The output of an **approved** entropy source consists of a status indication, and if the entropy source is operating correctly and entropy is available, a bitstring containing entropy is also provided. Otherwise, an error indication is returned as the status.

SP 800-90B discusses the handling of errors during the health testing of an entropy source. If the entropy source is unable to resolve the error, an error status indicator is returned to the calling application (e.g., the RBG routine calling the entropy source).

Each output from a properly functioning entropy source consists of a bitstring that has a fixed length, *ES_outlen*. This document requires the use of an **approved** entropy source with an assessed amount of entropy (*ES_entropy*) per *ES_outlen*-bit output that has been determined during implementation validation (see Section 12).

An interface to the entropy source is discussed in Section 7.4, and constructions for accessing an entropy source are provided in Section 10.3.

5.3.2 Live Entropy Source Availability

Three scenarios for the availability of an entropy source are considered in this document:

1) An entropy source is not available to fulfill requests,
2) An entropy source is available, but entropy cannot be immediately provided (e.g., because entropy is currently unavailable or collecting entropy is slow), or
3) An entropy source is available and entropy is immediately (or almost immediately) provided.

In cases 2 and 3, the entropy source is considered to be a Live Entropy Source: an **approved** entropy source that can provide the requested amount of entropy immediately or within an acceptable amount of time, as determined by the user or application requesting random bits from an RBG. Note that there is a distinction between the availability of an entropy source and the availability of entropy bits from an available entropy source. Also, note that entropy sources could only be available intermittently or during DRBG instantiation; entropy sources are considered to be Live only when actually available during requests for (pseudo) random bits.

A Live Entropy Source provides fresh entropy, which is required for an RBG to instantiate the initial DRBG in a DRBG chain or to provide prediction resistance. See Section 6 for a discussion of DRBG chains, and Sections 5.4 and 5.5.2 for discussions of prediction resistance.

A Live Entropy Source can be used to support any security strength using an appropriate construction as specified in this document. An NRBG always has a Live Entropy Source, so can support any security strength. However, this may not be the case for a DRBG (see Section 5.5).

A Live Entropy Source could be directly accessible (e.g., a DRBG has a Live Entropy Source that is always available), or it could be indirectly accessible via an RBG that has a Live Entropy Source (e.g., a DRBG can obtain entropy bits from an NRBG, which always has an available entropy source, or from another DRBG that has direct access to an entropy source).

5.3.3 Using a Single Entropy Source

A single entropy source may provide the required amount of entropy as a single bitstring, or multiple requests may be used to obtain the required amount of entropy. When multiple requests are needed, the entropy-source output can be concatenated, and the entropy in the resulting bitstring is the sum of the entropy contained in each component bitstring. See item 3 of Section 4.2 for additional information.

5.3.4 Using Multiple Entropy Sources

Entropy bitstrings may be obtained from multiple entropy sources. When multiple entropy sources are used, they **shall** be independent of each other. For one entropy source to be independent of another entropy source, the security boundaries of the entropy sources **shall not** overlap; the security boundary for an entropy source is declared during entropy-source validation.

When entropy bits are obtained from multiple independent entropy sources, the output bitstrings can be concatenated, and the entropy in the resulting bitstring is the sum of the entropy contained in each component bitstring. See item 4 of Section 4.2 for additional information.

5.3.5 External Conditioning

Conditioning may have been performed by an entropy source prior to providing output, but conditioning within the entropy source itself (i.e., internal conditioning) is not required by SP 800-90B. Whether or not entropy-source output was conditioned within the entropy source, the output of an entropy source could be conditioned prior to subsequent use by the RBG. Reasons for performing external conditioning might be to:

- Reduce the bias in the entropy-source output and distribute entropy across a bitstring,

- Reduce the length of the bitstring and compress the entropy into a smaller bitstring, and/or
- Ensure the availability of full-entropy bits.

Since this conditioning is done external to the entropy source, the entropy-source output is said to be *externally conditioned*.

An external conditioning function includes one or more iterations of a cryptographic algorithm that has been vetted for conditioning; such conditioning functions are listed or referenced in [SP 800-90B]. Section 10.3.2 provides further discussion on the use of external conditioning functions.

5.4 Prediction Resistance

An RBG may support prediction resistance, which means that a compromise of the internal state in the past or present will not compromise future RBG outputs. Prediction resistance may be provided automatically for all generation requests or may be provided on-demand, and requires the availability of a properly functioning Live Entropy Source to provide fresh entropy bits; if the entropy source fails, prediction resistance cannot be provided. The Live Entropy Source may be directly or indirectly accessible (see Section 5.3.2).

Properly functioning NRBGs compliant with SP 800-90C provide prediction resistance for each generation request because they always access a Live Entropy Source. Each call to the NRBG results in fresh entropy bits (see Section 5.6).

DRBGs with access to a Live Entropy Source can provide prediction resistance when requested to do so. Prediction resistance is accomplished by reseeding the DRBG using a randomness source that has access to a Live Entropy Source (e.g., an NRBG or a DRBG with access to a Live Entropy Source) and including a request for prediction resistance in the reseed request.

For a more complete discussion of prediction resistance, see SP 800-90A.

5.5 Deterministic Random Bit Generators (DRBGs)

5.5.1 General Discussion

An RBG could be a DRBG. A DRBG consists of a DRBG mechanism (i.e., an algorithm) and a randomness source; note that the difference between a DRBG and a DRBG mechanism is that the DRBG includes a randomness source, while the DRBG mechanism does not. A randomness source may be an entropy source that conforms to SP 800-90B, or an RBG that is ultimately based on an entropy source that conforms to SP 800-90B. Section 6 of this document (i.e., SP 800-90C) discusses randomness sources. Section 8 discusses the construction of a DRBG from a randomness source and a DRBG mechanism specified in SP 800-90A.

A DRBG **shall** be instantiated before it can provide pseudorandom bits using a randomness source that is available at that time. However, the randomness source may or may not be available after instantiation.

When the randomness source is a DRBG, this source DRBG **shall** not be the same DRBG instantiation as the DRBG being instantitated (i.e., the target DRBG) (see SP 800-90A).

5.5.2 Reseeding and Prediction Resistance

Applications using DRBGs may require that the DRBG be capable of periodically reseeding itself in order to thwart a possible compromise of the DRBG or to recover from an actual compromise.

The reseeding of a (target) DRBG requires the availability of a randomness source, either:

- An entropy source,
- A DRBG with or without access to an entropy source, or
- An NRBG (which has an entropy source).

If prediction resistance or guaranteed recovery from a compromise of the DRBG's internal state is desired, fresh entropy is needed, which requires the availability of a Live Entropy Source, i.e., in these cases, the randomness source for the (target) DRBG **shall** be either:

1. An entropy source,
2. An NRBG, or
3. A DRBG with access to a Live Entropy Source.

5.5.3 Security Strength Supported by a DRBG

A DRBG directly or indirectly supports a given security strength *s* if either:

- The DRBG has been instantiated at a security strength that is equal to or greater than *s*, or
- The DRBG has access to a Live Entropy Source (i.e., the DRBG's randomness source is a Live Entropy Source, an NRBG or one or more other DRBGs, one of which has access to a Live Entropy Source; see Section 6).

5.6 Non-deterministic Random Bit Generators (NRBGs)

An RBG could be an NRBG. An **approved** NRBG provides output bits that are indistinguishable from an ideal random sequence to any observer; that is, an NRBG provides full-entropy output – a request for *n* bits of output will result in a bitstring of *n* bits, with each bit providing one bit of entropy. See Section 9 for further discussions about NRBGs.

An NRBG is designed with access to a Live Entropy Source. Because an entropy source is always available, a properly functioning NRBG always provides fresh entropy and prediction resistance.

In addition to a Live Entropy Source, the NRBGs specified in this Recommendation include an **approved** DRBG mechanism. The NRBGs herein are constructed so that if the entropy source fails without detection, the security provided by the NRBG is reduced to the security strength of the **approved** DRBG used in the NRBG construction. This assumes that the DRBG has been properly instantiated with sufficient entropy to support that security strength.

6 Randomness Sources

In order to construct a DRBG or an NRBG that contains a DRBG mechanism, the RBG designer **shall** construct a source of secret, random or pseudorandom input for the DRBG mechanism, i.e., a randomness source. A randomness source is used by a DRBG mechanism to construct seed material for instantiation. It may also be used to construct seed material for reseeding automatically at the end of the reseed interval of the DRBG mechanism or for reseeding on demand, including fulfilling requests for prediction resistance.

There are two primary components that may be used to construct a randomness source: **approved** RBGs and **approved** entropy sources. A randomness source can, in fact, be a nested chain of RBGs (see Figure 3). In this figure, the inner RBGs in the "nest" (i.e., RBG 1 through RBG *n*-1) are considered to be higher-level RBGs than the target RBG (i.e., RBG *n*), and RBG 1 is the innermost or initial RBG in the chain. The entropy source used by RBG 1 is required for its instantiation, but may not be available after the instantiation of RBG 1.

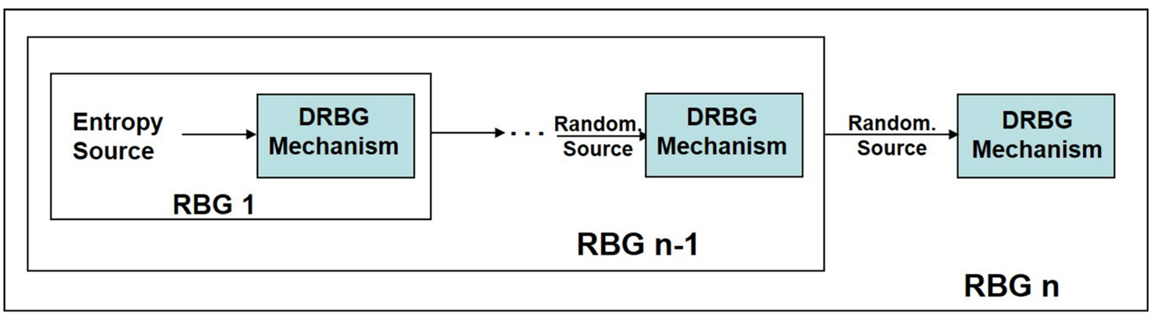

Figure 3: RBG Chain

To avoid possible confusion, a DRBG mechanism using a randomness source that will be accessed by a consuming application is called the *target DRBG mechanism*; a randomness source for the target DRBG that is an RBG, DRBG or NRBG is referred to as the *source RBG*, *source DRBG* or *source NRBG*, respectively. Note that the source RBG could be either a DRBG or an NRBG. A source DRBG may be implemented using the same DRBG design as the target DRBG (e.g., both the target and source DRBGs may be implemented as specified for an HMAC_DRBG using the same hash function), or may be implemented using different DRBG designs. When the target and source DRBG have the same design, they **shall** have different instantiations.

The target DRBG mechanism invokes a **Get_entropy_input** call, which includes the appropriate call for the selected randomness source (e.g., the **Get_entropy_input** call includes a **Generate_function** call if a DRBG is used as the randomness source and pseudorandom bits are requested). See Section 7 and Section 10 for further specifics about the **Get_entropy_input** call.

The requirements for the randomness source (s) are:

1. During instantiation, the randomness source(s) **shall** support at least the security strength that is intended for the target DRBG mechanism that is using it. Note that the maximum security strength that a target DRBG can support is limited by its design[2].
   a. A source RBG (i.e., a DRBG or NRBG) can be used to support the security strength to be provided by the constructed target DRBG under the following conditions:
      - If the source RBG is either 1) a DRBG with access to a Live Entropy Source or 2) an NRBG, then the target DRBG can be instantiated at any security strength when accessed as specified in this document. For example, if the desired security strength for the target DRBG is 256 bits, then a DRBG with a security strength of 128 bits can be used as the randomness source when it has access to a Live Entropy Source, and the appropriate constructions are used.
      - If the source RBG is a DRBG without a Live Entropy Source, then the target DRBG can be instantiated at a security strength that is less than or equal to the security strength of the source DRBG. For example, if the desired security strength for the target DRBG is 192 bits, then the (source) DRBG must have been instantiated at a security strength of at least 192 bits.
   b. An **approved** entropy source supports any desired security strength when used as a randomness source.
2. If the target DRBG is intended to allow reseeding, either on-demand or at the end of the DRBG's reseed interval, then the randomness source **shall** be available when the reseeding process is requested.
   a. A source DRBG with access to a Live Entropy Source or an NRBG can be used to reseed the target DRBG at any security strength when accessed as specified in this document.
   b. A source DRBG without a Live Entropy Source can be used to reseed the target DRBG at a security strength that is less than or equal to the security strength of the source DRBG.
   c. An **approved** entropy source can be used to reseed the target DRBG at any security strength.
3. If the target DRBG is intended to support requests for prediction resistance, then a Live Entropy Source **shall** be available in order to fulfill those requests. The randomness source for the target DRBG **shall** be either a source DRBG with access to a Live Entropy Source, an NRBG or an **approved** entropy source.

[2] For example, a DRBG using SHA-1 as a primitive can support security strengths of 112 and 128 bits, but cannot support security strengths of 192 and 256 bits.

4. If the target DRBG is not required to be reseeded or to support prediction resistance, then the randomness source is not required to be available after instantiation.

5. If the randomness source is not within the same sub-boundary as the target DRBG, then a secure channel **shall** be used to transfer data from the randomness source to the target DRBG (see Section 5.1).

6. If the **CTR_DRBG** is used as the target DRBG mechanism (see SP 800-90A), and a derivation function will not be used, then the randomness source used by the **CTR_DRBG shall** be:

   a) An NRBG,

   b) A DRBG with a Live Entropy Source that has been constructed to provide full-entropy output (see Section 10.4),

   c) An entropy source that has been assessed as providing full-entropy output, or

   d) An entropy source and external conditioning function that are used together to provide full-entropy output (see Section 10.3.3.3).

7 RBG Interfaces

Functions used within this document for accessing DRBGs, NRBGs and entropy sources are provided below. Each function uses one or more of the input parameters listed for that function during its execution, and **shall** return a status code that **should** be checked by the consuming application.

If the status code indicates a *success*, then additional information may also be returned, such as a state handle from an instantiate call or the bits that were requested to be generated during a generate call.

If the status code indicates a *failure* of an RBG component, then see Section 12.1.3 for error-handling guidance.

The status code may also indicate other conditions, but this is not required. Examples include:

- The lack of a Live Entropy Source when prediction resistance is requested (an appropriate response would be to notify the consuming application of the problem and deny the request), and
- The current unavailability of entropy bits from an available entropy source (an appropriate response might be to re-issue the request at a later time).

Note that if the status code does not indicate a success, a null string **shall** be returned with the status code if information other than the status code could be returned.

7.1 General Pseudocode Conventions

All algorithms in SP 800-90C are described in pseudocode that is intended to explain the algorithm's function. These pseudocode conventions are not intended to constrain real-world implementations, but to provide a consistent notation to describe the constructions herein. By convention, unless otherwise specified, integers are 32-bit unsigned, and when used as bitstrings, they are represented in big-endian format.

7.2 DRBG Function Calls

7.2.1 Basic DRBG Functions

A DRBG contains a DRBG mechanism and a randomness source. See SP 800-90A for more information about DRBG mechanisms, and Section 6 for randomness sources. Note that, in some situations, not all input parameters for a function are required, and not all output information is returned. The DRBG supports the following interfaces:

1. (*status, state_handle*) = **Instantiate_function**(*requested_instantiation_security_strength, prediction_resistance_flag, personalization_string*).

   The **Instantiate_function** is used to instantiate a DRBG at a requested security strength using a randomness source and an optional personalization string; the function call could also indicate whether the DRBG will need to provide prediction resistance. The randomness source is accessed by the **Instantiate_function** using a **Get_entropy_input** call (see item 4 below). If the returned status code for the

Instantiate_function indicates a success, a state handle will be returned to indicate the particular DRBG instance; the state handle will be used in subsequent calls to the DRBG (e.g., during a **Generate_function** call). If the status code indicates an error, a Null state handle will be returned.

2. (*status*, *returned_bits*) = **Generate_function**(*state_handle*, *requested_number_of_bits*, *requested_security_strength*, *prediction_resistance_request*, *additional_input*).

   The **Generate_function** requests that a DRBG generate a specified number of bits. The request indicates the DRBG instance to be used (using the state handle returned by an **Instantiate_function** call), the number of bits to be returned, the security strength that the DRBG must support and whether or not prediction resistance is to be invoked during this execution of the **Generate_function**. Optional additional input may also be incorporated into the function call. If the returned status code indicates a success, a bitstring containing the newly generated bits is returned. If the status code indicates an error, the *returned_bits* will consist of a Null string.

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

   The **Reseed_function** is optional in a DRBG. When present, it is used to acquire new entropy input for the DRBG instance indicated by the state handle. The call may indicate a requirement for the use of a Live Entropy Source during the reseeding process (via the *prediction_resistance_request* parameter), and optional additional input may be incorporated into the process. The **Reseed_function** obtains the entropy input from a randomness source using a **Get_entropy_input** call (see item 4). An indication of the status is returned.

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

   The **Get_entropy_input** call is performed within the instantiate and reseed functions (items 1 and 3 above) to access a randomness source. The specifics of the call depend on the randomness source to be used; constructions for the **Get_entropy_input** function are provided in Section 10. In general, the call indicates (at a minimum) the minimum amount of entropy to be returned. The call may also include the minimum and/or maximum length of the bitstring to be returned, as well as a request that prediction resistance be provided (i.e., a Live Entropy Source is required). If the returned status code indicates success, a bitstring containing the requested entropy input is also returned. If the status code indicates an error, the *entropy_input* will be a Null string.

Note that the use of the **Uninstantiate_function** specified in SP 800-90A is not explicitly discussed in SP 800-90C.

7.2.2 Additional DRBG Function

An additional DRBG function is included in this document in order to allow a DRBG to provide full-entropy output upon request. If a DRBG has access to a Live Entropy Source, it can provide prediction resistance and full-entropy output using the construction in Section 10.4. The following function call is provided for this purpose:

(*status*, *returned_bits*) = **General_DRBG_Generate**(*state_handle*, *requested_number_of_bits*, *security_strength*, *full_entropy_request*, *prediction_resistance_request*, *additional input*).

This function call is especially useful for the case where the target DRBG's randomness source does not provide full-entropy itself (i.e., the randomness source is a DRBG with access to a Live Entropy Source, or an entropy source without an external conditioning function to condition the entropy-source output to provide full entropy). For randomness sources that inherently provide full entropy (e.g., an NRBG or an entropy source that provides full-entropy output), the **DRBG_Generate** function call in Section 10.2.1 may be more efficient.

7.3 NRBG Function Calls

A non-deterministic random bit generator (NRBG) supports the following interfaces. The definition of the parameters used as input and output are the same as those used for the DRBG function calls in Section 7.2.

1. (*status*, *state_handle*) = **NRBG_ Instantiate**(*prediction_resistance_flag*, *personalization_string*).

   The **NRBG_ Instantiate** function is used to instantiate the DRBG mechanism within the NRBG; this will result in a call to the **Instantiate_function** provided in Section 7.2 and SP 800-90A. A prediction-resistance capability may be requested for the DRBG instantiation, and a personalization string may be provided for use during the DRBG instantiation process. If the returned status code indicates success, a state handle will be returned to indicate the particular DRBG instance that is to be used by the NRBG; the state handle will be used in subsequent calls to that DRBG (e.g., during an **NRBG_Generate** call). If the status code indicates an error, a Null state handle will be returned.

2. (*status, returned_bits*) = **NRBG_Generate**(*state_handle*, *requested_number_of_bits*, *additional_input*).

   The **NRBG_Generate** function is used to request full-entropy output from an NRBG; this function results in calls to the entropy source and to the DRBG mechanism used by that NRBG. This call accesses the DRBG mechanism using the **Generate_function** call provided in Section 7.2 and SP 800-90A, and the input parameters in the **NRBG_Generate** call are used when calling that DRBG. If the returned status code indicates success, a bitstring containing the newly generated bits is returned. If the status code indicates an error, the *returned_bits* will be a Null string.

3. (*status*, *returned_bits*) = **NRBG_DRBG_Generate**(*state_handle*, *requested_number_of_bits*, *requested_security_strength*, *prediction_resistance_request*, *additional_input*).

   An **NRBG_DRBG_Generate** function may optionally be used to directly access the DRBG instantiation associated with the NRBG to request the generation of a specified number of bits. This function calls the DRBG mechanism using the **Generate_function** call provided in Section 7.2 and SP 800-90A, optionally requesting prediction resistance from the DRBG and using the input parameters provided to the

NRBG_DRBG_Generate call. If the returned status code indicates success, a bitstring containing the requested bits is returned.

7.4 Entropy Source Calls

An entropy source, as discussed in SP 800-90B, is a mechanism for producing bitstrings that cannot be completely predicted, and whose unpredictability can be quantified in terms of min-entropy. This Recommendation allows the use of either a single entropy source or multiple independent entropy sources. The interface routine to an entropy source is accomplished using the following call.

(*status*, *entropy_bitstring*) = **Get_Entropy**(*requested_entropy*, *max_length*),

where *max_length* is an optional parameter that indicates the maximum length allowed for *entropy_bitstring*.

The **Get_Entropy** interface function is responsible for obtaining entropy from the entropy source(s) in whatever manner is required (e.g., by polling the entropy source(s) or extracting bits containing entropy from a pool of bits collected as the result of system interrupts). An RBG implementer is responsible for the particulars of the actual interaction with the entropy source(s) in the function, but some guidance is provided in Section 10.3.1.

The **Get_Entropy** function is invoked from one of the **Get_entropy_input** constructions specified in Section 10.3.3.

7.5 Conditioning Function Calls

The output of an entropy source may be externally conditioned using vetted methods prior to subsequent use by the RBG. These methods are based on the use of **approved** hash functions or **approved** block-cipher algorithms. The use of conditioning is discussed in Section 10.3.2.

For the hash functions or block-cipher algorithms, the conditioning function calls include a string of bits (*entropy_bitstring*) obtained from one or more calls to the entropy source.

Some of the algorithms also include a *Key* as input; this key is also discussed in Section 10.3.2.1. The key **shall** be available prior to invoking the algorithm.

7.5.1 Conditioning Functions Based on Approved Hash Functions

Conditioning functions may be based on the use of **approved** hash functions and may include optional additional data (denoted as *A*) to be hashed with the entropy bits (denoted as *entropy_string*). In this case, the conditioning function includes one of the following calls:

1. Using an **approved** hash function directly: The conditioning function makes the following call to the hash function:

   output_string = **Hash**(*entropy_string* || *A*).

   The length of the *output_string* is equal to the length of the output block of the selected hash function.

2. Using HMAC with an **approved** hash function: The conditioning function makes the following call to HMAC:

   output_string = **HMAC**(*Key*, *entropy_string* || *A*).

The length of the *output_string* is equal to the length of the output block of the selected hash function.

3. Using an **approved** hash function in the hash-based derivation function specified in SP 800-90A: The conditioning function makes the following call:

   (*status*, *requested_bits*) = **Hash_df**(*entropy_string* || *A*, *no_of_bits_to_return*).

   The derivation function operates on the provided input string (*entropy_string* || *A*) and, if no error is indicated by the returned *status*, a bitstring of the requested number of bits is returned.

7.5.2 Conditioning Functions Based on Approved Block-Cipher Algorithms

Conditioning functions may be based on the use of **approved** block-cipher algorithms and may include optional additional data (denoted as *A*) to be concatenated to the entropy bits (denoted as *entropy_string*). In this case, the conditioning function includes one of the following calls:

1. Using CMAC with an **approved** block-cipher algorithm as specified in SP 800-38B. The conditioning function makes the following call:

   output_string = **CMAC**(*Key*, *entropy_string* || *A*).

   The length of the *output_string* is equal to the length of the output block of the selected block-cipher algorithm. Note that a key **shall** be available prior to invoking CMAC.

2. Using CBC-MAC with an **approved** block-cipher algorithm as specified in Appendix C. The conditioning function makes the following call:

   output_string = **CBC-MAC**(*Key*, *entropy_string* || *A*).

   The length of the *output_string* is equal to the length of the output block of the selected block-cipher algorithm. The length of *entropy_string* **shall** be an integer multiple of the block length, and all uses of CBC-MAC in an RBG **shall** have the same fixed length for *enropy_bitstring*. The key **shall** be available prior to invoking CMAC.

3. Using an **approved** block-cipher algorithm in a derivation function as defined in SP 800-90A. The conditioning function makes the following call:

   (*status*, *requested_bits*) = **Block_Cipher_df**(*entropy_string* || *A*, *no_of_bits_to_return*).

   The derivation function operates on the provided input string (*entropy_string* || *A*) and, if no error is indicated by the returned *status*, a bitstring of the requested number of bits is returned. If an error is indicated by the status code, then *requested_bits* is the Null string. The input string **shall** be a multiple of eight bits in length, and be no longer than 512 bits in length. Note that the key for this algorithm is defined within the **Block_Cipher_df** specification.

8 DRBG Construction

A DRBG is constructed from a DRBG mechanism and a randomness source. DRBG mechanisms are specified in SP 800-90A, and examples of DRBGs are provided in Appendix A.

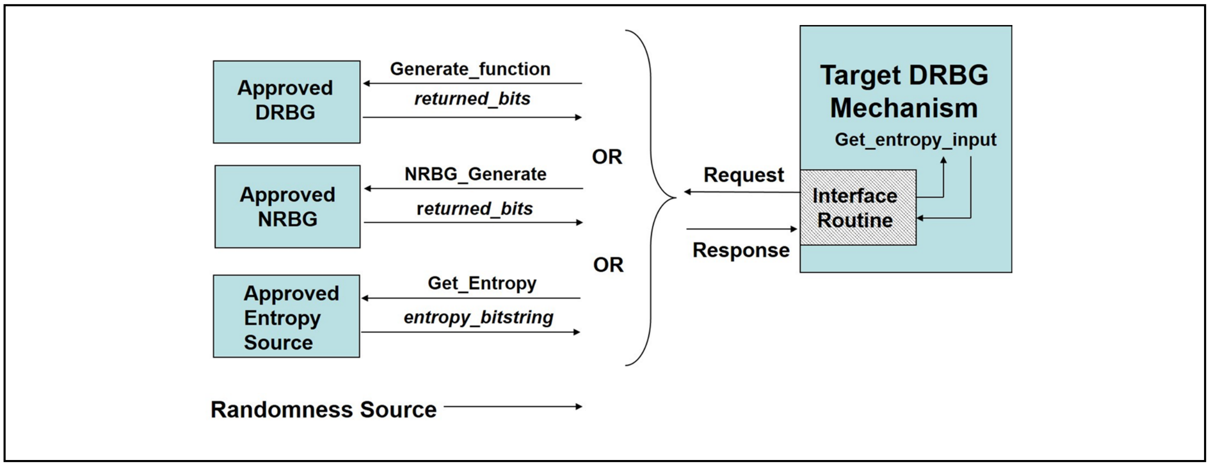

Figure 4: Randomness Sources for a DRBG

As shown in Figure 4, the randomness source for a target DRBG could be an **approved** DRBG, an **approved** NRBG or an **approved** entropy source. Note that the function calls and returned results are depicted.

A source DRBG could be a chain of **approved** DRBGs (see Section 6), consisting of a target DRBG and one or more higher-level DRBGs that serve as the source for the target DRBG. Section 10 of this document provides constructions to access the appropriate randomness source from the DRBG's **Get_entropy_input** call.

8.1 DRBG Functionality Depending on Randomness Source Availability

A randomness source **shall** be available for DRBG instantiation, but need not be available thereafter; however, if reseeding is to be performed, then a randomness source **shall** be available for the reseeding operation. The randomness source is either an entropy source, an NRBG or a source DRBG (with or without access to a Live Entropy Source). If the reseeding operation is used to provide prediction resistance, fresh entropy is required, and a source DRBG used for reseeding **shall** have access to a Live Entropy Source. Table 1 summarizes the availability of randomness sources and entropy, and indicates the possible DRBG functionality.

Table 1: DRBG Functionality

	Randomness Source Availability	Live Entropy Source?	Comments
1	Whenever required	Yes	A Live Entropy Source is always available; the randomness source is an entropy source, an NRBG, or a source DRBG with access to a Live Entropy Source. A target DRBG can be instantiated, generate bits, be reseeded, and provide prediction resistance.
2	Whenever required	No	A randomness source is always available; in this case, the randomness source is a source DRBG with no access to a Live Entropy Source. A target DRBG can be instantiated, generate bits, and be reseeded, but cannot provide prediction resistance.
3	During instantiation only	No	A randomness source is only available for instantiation; the randomness source is an entropy source, an NRBG, or a source DRBG with or without access to a Live Entropy Source. A target DRBG can be instantiated and generate bits, but cannot be reseeded or provide prediction resistance.
4	Intermittently	Yes	A Live Entropy Source is available only intermittently; the randomness source is an entropy source, an NRBG, or a source DRBG with access to a Live Entropy Source. A target DRBG can be instantiated and generate bits, but reseeding, including providing prediction resistance, can only be done when the randomness source is available.
5	Intermittently	No	A randomness source is available intermittently; the randomness source is a source DRBG with no access to a Live Entropy Source. The target DRBG can be instantiated and generate bits, but can be reseeded only when the randomness source is available. Prediction resistance cannot be provided.

When a source DRBG is used as a randomness source, its use for instantiating and reseeding a target DRBG is subject to the restrictions discussed in Section 6.

If prediction resistance is requested, and a Live Entropy Source is not available (e.g., the entropy source indicates that it has failed or entropy output is not currently available), the consuming application **shall** be notified, and output other than the status **shall not** be returned for that request.

When a source DRBG is used to instantiate or reseed a target DRBG, the target and source DRBG instantiations **shall not** be the same.

Sections 8.2 - 8.4 further address the differences provided by the use or non-use of Live Entropy Sources.

8.2 DRBG Instantiation

A target DRBG is instantiated using a randomness source and the **Instantiate_function** (see Section 7.2 and in SP 800-90A). This function uses a **Get_entropy_input** call to obtain entropy input from the randomness source. Section 10 contains several constructions for this function. The construction to be used for the **Get_entropy_input** function is selected as follows:

1. If the randomness source is a source DRBG, the DRBG may or may not have access to a Live Entropy Source. During the instantiation of the target DRBG:
    a. If the source DRBG has access to a Live Entropy Source, either the **Get_entropy_input** construction in Section 10.1.1 or Section 10.1.2 **shall** be used. However, if the security strength of the target DRBG is intended to be higher than the security strength of the source DRBG, then the construction in Section 10.1.2 **shall** be used.
    b. If the source DRBG does not have access to a Live Entropy Source, the **Get_entropy_input** construction in Section 10.1.1 **shall** be used. Note that an error will be returned if the security strength indicated in the **Get_entropy_input** call is greater than the security strength instantiated for the source DRBG.
2. If the randomness source is a source NRBG, the **Get_entropy_input** construction in Section 10.2 **shall** be used.
3. If the randomness source is an entropy source, a **Get_entropy_input** construction in Section 10.3.3 **shall** be used.

Note that in some cases, prediction resistance can be requested for the instantiation during the **Instantiate_function** call; if an entropy source does not appear to be available during the execution of this function (as in case 1.b above) or will not be available during normal operation, then an error indicator **shall** be returned to the consuming application.

Also, recall that the security strength for a DRBG is set during the instantiation process, and is recorded in the internal state for that instantiation (see SP 800-90A).

8.3 Generation of Output Using a DRBG

A consuming application requests that a target DRBG generate pseudorandom output using the **Generate_function** specified in Section 7.2 and SP 800-90A.

During the execution of the **Generate_function**, an implementation may determine that reseeding is required (i.e., the end of the reseed interval has been reached – see SP 800-90A). Reseeding requires the availability of a randomness source (see Section 6). If a randomness source is not available when reseeding is required, then an error indication **shall** be returned to the consuming application. Otherwise, a request for reseeding is made (see Section 8.4); this request may or may not include a request for prediction resistance.

If prediction resistance is requested during a **Generate_function** call to obtain fresh entropy for the DRBG, and 1) prediction resistance was not requested during the successful instantiation of the DRBG, or 2) if a Live Entropy Source is not currently available, then an error indicator **shall** be returned to the consuming application. Otherwise, a request for reseeding is made with prediction resistance requested to indicate that access to a Live Entropy Source is required during the execution of the reseed function (see Section 8.4).

A target DRBG with access to a Live Entropy Source may provide full-entropy output when the construction in Section 10.4 is used. In this case, the DRBG is requested to provide *s*/2 bits of output with prediction resistance, where *s* is the security strength of the DRBG instantiation. Successive calls to the DRBG are required to obtain a (cumulative) bitstring longer than *s*/2 bits. Note that this capability can be considered as an ad-hoc Oversampling NRBG.

8.4 DRBG Reseeding

A target DRBG may be reseeded as a result of 1) a reseeding request by a consuming application, 2) in response to a request for prediction resistance during the execution of a **Generate_function** request (see Section 8.3), or 3) as otherwise determined during the **Generate_function** execution (e.g., the end of the reseed interval has been reached) (see Section 8.3). The call for the reseed function is included in Section 7.2. This function uses a **Get_entropy_input** call to obtain entropy input for the target DRBG.

Reseeding of the target DRBG proceeds as follows:

1. If a randomness source is not available when reseeding of the target DRBG is requested, then an error indication **shall** be returned to the consuming application (see SP 800-90A).
2. If prediction resistance is requested, and a Live Entropy Source is not available, then an error indication **shall** be returned to the consuming application (see SP 800-90A)
3. If a randomness source returns an indication that entropy is not currently available, then this indication **shall** be provided to the consuming application.
4. If the randomness source is a source DRBG, and a Live Entropy Source is available:
    - If prediction resistance has been requested, and the security strength of the target DRBG does not exceed the security strength of the source DRBG, then the **Get_entropy_input** construction in either Section 10.1.1 or Section 10.1.2 **shall** be used.
    - If prediction resistance has been requested, and the security strength of the target DRBG is higher than the security strength of the source DRBG, then the construction in Section 10.1.2 **shall** be used.
    - If prediction resistance has not been requested, then the **Get_entropy_input** construction in either Section 10.1.1 or 10.1.2 **shall** be used.
5. If the randomness source is a source DRBG and a Live Entropy Source is not available:
    - If the security strength of the target DRBG exceeds the security strength of the source DRBG, then an error indication **shall** be returned to the consuming application.
    - If the security strength of the target DRBG does not exceed the security strength of the source DRBG, then the **Get_entropy_input** construction in either Section 10.1.1 or 10.1.2 **shall** be used.

6. If the randomness source is a (source) NRBG, the **Get_entropy_input** construction in Section 10.2 **shall** be used.
7. If the randomness source is an entropy source, a **Get_entropy_input** construction in Section 10.3.3 **shall** be used.

8.5 Sources of Other DRBG Inputs

Fully implementing a DRBG requires a decision about the inclusion of nonces, personalization strings, and additional input, as well as how this information will be obtained.

1. Nonces: In the case of the nonces specified in SP 800-90A, if a nonce is required and the nonce is not provided by the implementation environment (e.g., using a clock and/or a counter), then it **shall** be provided by the randomness source. See SP 800-90A for further discussion.
2. Personalization strings: Personalization strings are optional input parameters that may be used during DRBG instantiation to differentiate between instantiations. If possible, the DRBG implementation **should** allow the use of a personalization string. Details on personalization strings are provided in SP 800-90A.
3. Additional input: SP 800-90A allows additional input to be provided by a consuming application during the **Generate_function** and **Reseed_function** requests. RBG designers **should** include this option in the selected DRBG mechanism. This input could, for example, include information particular to a request for generation or reseeding, or could contain entropy collected during system activity.

9 NRBG Constructions

An NRBG produces bits with full entropy. These bits are expected to be indistinguishable (in practice) from an ideal random sequence to any adversary. As stated in Section 5.6, this document provides constructions for NRBGs. The following two constructions are provided:

- XOR Construction – This NRBG construction is based on combining the output of an **approved** entropy source with the output of an instantiated, **approved** DRBG using an exclusive-or (XOR) operation (see Section 9.3).
- Oversampling Construction – This NRBG is based on using an **approved** entropy source that provides entropy input for an **approved** DRBG (see Section 9.4).

The advantages of using these NRBGs include the following:

- If the underlying DRBG mechanism in the NRBG has been instantiated securely, and the entropy source fails in an undetected manner, the NRBG will continue to provide random outputs, but at the security strength of the DRBG instantiation (the "fall-back" security strength), rather than providing outputs with full entropy.
- Small deviations in the behavior of the entropy source in an NRBG will be masked by the DRBG output.

In both NRBG constructions, an entropy source that deviates just slightly from its correct behavior leads to a very small security impact; the DRBG mechanisms mask any misbehavior, and an adversary who cannot break the DRBG mechanism's security will not be able to detect the misbehavior. When the entropy source malfunctions slightly, an adversary who can break the DRBG mechanism has only a slightly better chance to distinguish the NRBG outputs from ideal random outputs than he would if the entropy source is operating correctly.

Examples of NRBGs are provided in Appendices A.1 and A.2.

9.1 Entropy Source Access and General NRBG Operation

Upon the receipt of a request for random bits from a consuming application, an NRBG will need to access its entropy source(s) to obtain one or more bitstrings with entropy. The entropy source(s) could 1) (almost) immediately return the requested output, 2) delay its response to the request until entropy is available, 3) return an explicit indication that sufficient entropy is not yet available, or 4) return an indication of an error.

The details of interaction with the entropy source are the responsibility of the implementer of the entropy-source call discussed in Section 7.4. This function may need to access the entropy source(s) several times in order to obtain sufficient entropy to fulfill the **Get_Entropy** request. Section 5.3.4 discusses the entropy that results when the output of multiple entropy sources is used to obtain the requested entropy. If multiple entropy sources are used, and at least one of these has not failed, then NRBG operations may continue using the remaining (non-failed) entropy sources. Additional guidance for accessing the entropy source is provided in Section 10.3.1.

After the entropy source(s) provides its output, the NRBG may perform external conditioning. Further discussion on the use of external conditioning is provided in Section 10.3.2. The NRBG then uses the resulting bitsting as specified for each NRBG construction below (see Sections 9.3 and 9.4).

9.2 The DRBG Mechanism within the NRBG

In the NRBG constructions specified in Sections 9.3 and 9.4, the DRBG instantiation used by the NRBG **shall** be instantiated at the highest possible security strength that is consistent with its cryptographic components and the security strengths supported by this Recommendation (i.e., either 112, 128, 192, or 256 bits).

The DRBG mechanism included in the NRBG may be implemented to be directly accessible by a consuming application. Direct requests to the DRBG mechanism may use either the same DRBG instantiation used by the NRBG, or a separate instantiation may be used. The DRBG instantiation(s) **shall** be used as discussed in Section 8, including any prediction resistance capability.

If a separate instantiation of the DRBG used by the NRBG is used for direct DRBG access, the separate instantiation may have any security strength supported by the DRBG's cryptographic components and this Recommendation, rather than at the highest security strength, as required by the NRBG construction. For example, a DRBG based on SHA-1 could be instantiated at 128 bits for the instantiation used for the NRBG, and at 112 bits for the instantiation used for direct access. When a separate instantiation of the DRBG is used, the randomness source for that DRBG instantiation may be any randomness source discussed in Section 6, including the entropy source of the NRBG.

9.3 XOR-NRBG Construction

The XOR-NRBG construction is shown in Figure 5; an example is provided in Appendix A.1.

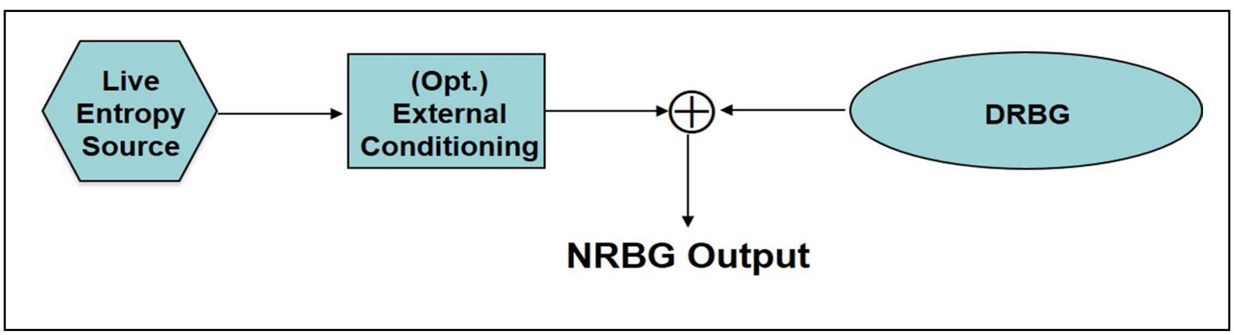

Figure 5: XOR-NRBG Construction

For the XOR-NRBG construction:

- One or more Live Entropy Sources **shall** be used. The input to the exclusive-OR function above **shall** be one of the following:
    - An **approved** entropy source as specified in SP 800-90B that provides full-entropy output,
    - An **approved** entropy source that is externally conditioned as specified in Section 10.3.2 to provide full-entropy output,

    - Multiple **approved** independent entropy sources whose outputs are combined and conditioned as specified in Section 10.3.2 to provide full-entropy output, or
    - An NRBG designed as specified for the Oversampling Construction (see Section 9.4).
- A DRBG that accesses a randomness source for instantiation **shall** be used (see Section 6). The randomness source need not be the entropy source used by the NRBG. Note that the DRBG mechanism is subject to the normal reseeding requirements of a DRBG. If the reseeding of the DRBG is required (e.g., because the DRBG may reach the end of its reseed interval), then the DRBG **shall** also incorporate a **Reseed_function**.
- The bits from the randomness source that are used as input to the DRBG (e.g., to instantiate or reseed the DRBG) **shall not** be used for any other purpose (e.g., as bits within the NRBG construction that are XORed with the output of the DRBG to produce the NRBG output for a consuming application)[3].
- During NRBG requests to generate random bits, the DRBG is not requested to provide prediction resistance. Note, however, that the DRBG could provide prediction resistance when accessed directly.

9.3.1 Instantiation of the DRBG used by the XOR-NRBG

The DRBG instantiation used in the XOR-NRBG **shall** be instantiated at its highest security strength. Let *highest_DRBG_security_strength* be the highest security strength that the DRBG mechanism can assume (see SP 800-90A for this value).

NRBG_ Instantiate:

Input: integer *prediction_resistance_flag*, string *personalization_string*.

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

Process:

1. (*status*, *state_handle*) = **Instantiate_function**(*highest_DRBG_security_strength, prediction_resistance_flag*, *personalization_string*).
2. Return (*status*, *state_handle*).

Step 1 instantiates the DRBG at its highest-possible security strength. The *prediction_resistance_flag* and *personalization_string* are optional parameters to the **NRBG_ Instantiate** call; if provided, they **shall** be passed to the DRBG's **Instantiate_function**. Note that the **Instantiate_function** accesses its randomness source using a **Get_entropy_input** call; Section 8.2 discusses the **Get_entropy_input** call for instantiating the DRBG.

In step 2, the value of *status* and *state_handle* returned in step 1 are returned to the consuming application; note that if the *status* does not indicate a successful instantiate process (i.e., an error

[3] This follows the general rule that bits conaining entropy must only be used once. Thus, entropy bits used to seed or reseed the DRBG, and entropy-source output to be XORed into the DRBG outputs for this construction must not be reused.

is indicated), the *state_handle* will be invalid. The handling of status codes by the consuming application is discussed in Section 7.

9.3.2 XOR-NRBG Generation

Let *highest_DRBG_security_strength* be the highest security strength that the DRBG mechanism can assume, let *n* be the requested number of bits, and let the *state_handle* be the value returned from the **NRBG_ Instantiate function** (see Section 9.3.1).

NRBG_Generate:

Input: integer (*state_handle*, *n*), string *additional_input*.

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

Process:

1. (*status*, *ES_bits*) = **Get_entropy_input**(*n*, *n*, *n*).
2. If (*status* ≠ SUCCESS), then return (*status*, Null).
3. (*status*, *DRBG_bits*) = **Generate_function**(*state_handle*, *n*, *highest_DRBG_security_strength*, *additional_input*).
4. If (*status* ≠ SUCCESS), then return (*status*, *Null*).
5. *returned_bits* = *ES_bits* ⊕ *DRBG_bits.*
6. Return (SUCCESS, *returned_bits*).

Step 1 requests that the entropy source generate bits. Since full-entropy bits are required, the **Get_entropy_input** construction in Section 10.3.3.1 **shall** be used if the entropy source provides full-entropy output; otherwise, the construction in Section 10.3.3.3 **shall** be used to condition the entropy-source output to obtain full-entropy bits. If the request is not successful, abort the **NRBG_Generate** function, returning the *status* received in step 1 and a Null string as the *returned_bits* (see step 2). If *status* indicates a success, *ES_bits* contains the entropy bits to be used later in step 5.

In step 3, the DRBG is requested to generate bits at its highest security strength. If additional input is provided in the **NRBG_Generate** call, it **shall** be included in the **Generate_function** call. Note that in the **NRBG_Generate** call, the NRBG's DRBG instantiation is not requested to provide prediction resistance. If the request is not successful, the **NRBG_Generate** function is aborted, and the *status* received in step 3 and a Null string are returned to the consuming application (see step 4). If *status* indicates a success, *DRBG_bits* contains the pseudorandom bits to be used in step 5.

Note that it is possible that the DRBG would require reseeding during the **Generate_function** call in step 3. If a reseed of the DRBG mechanism is required during NRBG generation, it **shall** use the **DRBG_Reseed** function (see Section 7.2).

Step 5 combines the bitstrings returned from the entropy source and the DRBG using an XOR operation; the resulting bitstring is returned to the consuming application in step 6.

9.3.3 Direct DRBG Access

The DRBG mechanism may be directly accessed as a DRBG using the same or a different instantiation than that used when the DRBG mechanism is performing as part of the NRBG.

If the DRBG instantiation is different than the DRBG instantiation used by the XOR-NRBG (i.e., the same DRBG mechanism is used but with a different internal state), then access to the DRBG is discussed in Section 8).

If the directly accessed DRBG instantiation is the same as the instantiation used for the NRBG, then the **NRBG_DRBG Generate** call specified in Section 7.3 is used (see below).

NRBG_DRBG_Generate:

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

Output: integer *status*, bitstring *returned_bits*.

Process:

1. (*status*, *returned_bits*) = **Generate_function** (*state_handle*, *requested_number_of_bits*, *requested_security_strength*, *prediction_resistance_request*, *additional_input*).
2. Return *status*, *returned_bits*.

In step 1, the NRBG's DRBG instantiation is requested to generate bits; the input parameters provided in the **NRBG_DRBG_Generate** call are provided to the DRBG in the **Generate_function** call. Note that prediction resistance can be requested, unlike the **Generate_function** request in accessing the NRBG (see Section 9.3.2). The returned *status* code and bitstring (i.e., *returned_bits*) are returned to the consuming application in step 2. Note that *returned_bits* will be the *Null* string if the status does not indicate a success.

When reseeding is required during the generate request (i.e., because prediction resistance is requested or the DRBG instantiation has reached the end of its reseed interval), the **Reseed_function** specified in Section 7.2 and SP 800-90A **shall** be used. The randomness source used by the **Reseed_function** may be any of those discussed in Section 6, including the entropy source of the NRBG.

9.4 The Oversampling-NRBG Construction

The Oversampling-NRBG construction is shown in Figure 6, and an example is provided in Appendix A.2. The DRBG mechanism within the NRBG repeatedly accesses a Live Entropy Source to obtain prediction resistance (i.e., reseeding the DRBG from the entropy source with sufficient entropy bits for the instantiated security strength of the DRBG mechanism). External conditioning of the entropy-source output may optionally be performed. In this NRBG construction, multiple calls requesting prediction resistance are made to the DRBG until the number of bits requested by the NRBG's consuming application have been obtained. In each DRBG call, a bitstring whose length is equal to half the security strength of the DRBG instantiation is requested and returned. This results in full-entropy outputs.

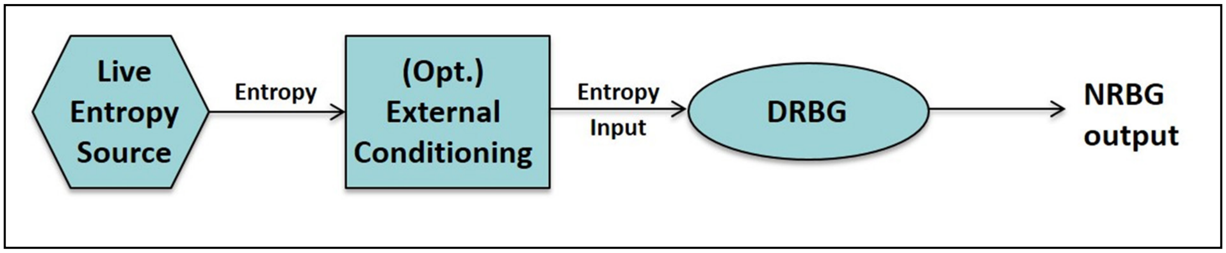

Figure 6: Oversampling-NRBG Construction

The security argument is as follows: if the Live Entropy Source is functioning correctly, the outputs of the DRBG are affected by the fresh entropy provided by the Live Entropy Source and the accumulated entropy from the DRBG instantiation and previous calls to the Live Entropy Source. If there is an undetected failure in the Live Entropy Source, the DRBG mechanism will continue to function as a DRBG, using whatever entropy has been inserted into the DRBG prior to the failure.

For the Oversampling-NRBG construction:

- A Live Entropy Source **shall** be used,
- Optional external conditioning may be performed, and
- A DRBG mechanism with a prediction resistance capability **shall** be used that results in a reseed of the DRBG for each request for bits in the NRBG construction. This means that the DRBG **shall** include a reseed function.

9.4.1 Instantiation of the DRBG used by the Oversampling NRBG

The DRBG instantiation used by the Oversampling NRBG **shall** be instantiated at its highest security strength. Let *highest_DRBG_security_strength* be the highest security strength that the DRBG mechanism can assume (see SP 800-90A).

NRBG_ Instantiate:

Input: string *personalization_string*.

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

Process:

1. (*status*, *state_handle*) = **Instantiate_function**(*highest_DRBG_security_strength*, *prediction_resistance _flag* = TRUE, *personalization_string*).
2. Return (*status*, *state_handle*).

Step 1 instantiates the DRBG at its highest-possible security strength using the **Instantiate_function** call (see Section 7.2 and SP 800-90A). Since prediction resistance is required for this NRBG construction, the *prediction_resistance_flag* **shall** be set to TRUE. A *personalization_string* is an optional parameter, but **shall** be used if it is provided in the **NRBG_ Instantiate** call. Note that the **Instantiate_function** accesses its randomness source using a **Get_entropy_input** call; Section 8.2 discusses the **Get_entropy_input** call for instantiating the DRBG.

In step 2, the value of *status* and *state_handle* returned in step 1 are returned to the consuming application; note that if the *status* does not indicate a successful instantiate process (i.e., an error is indicated), the *state_handle* will be invalid. The handling of status codes by the consuming application is discussed in Section 7.

9.4.2 Oversampling-NRBG Generation

Let *n* be the requested number of bits, let *state_handle* be the value returned from the **NRBG_ Instantiate** function (see Section 9.4.1) and let *s* be the *highest_DRBG_security_strength* (as used in Section 9.4.1).

NRBG_ Generate:

Input: integer (*state_handle*, *n*), string *additional_input*.

Output: integer *status*, bitstring *returned_bits*.

Process:

1. *tmp* =*Null*.
2. *sum* = *0*.
3. While (*sum* < *n*)
   3.1 (*status*, *returned_bits*) = **Generate_function**(*state_handle*, *s*/2, *s*, *prediction_resistance_request* = TRUE, *additional_input*).
   3.2 If (*status* ≠ SUCCESS*)*, then return (*status*, *Null*).
   3.3 *tmp* = *tmp* || *returned_bits.*
   3.4 *sum* = *sum* + *s/*2.
4. Return (SUCCESS, **leftmost**(*tmp*, *n*)).

The bitstring intended to collect generated bits for return to the calling application (i.e., *tmp*) is initialized to the null bitstring in step 1, and a counter for recording the amount of entropy obtained is initialized to zero in step 2.

In step 3, the DRBG is requested to generate bits until the requested number of full-entropy bits is accumulated.

In step 3.1, the DRBG is requested to generate bits with prediction resistance (i.e., *prediction_resistance_request* is set to TRUE). For each call to the **Generate_function**, *s*/2 bits of output are requested from the DRBG, which provides *s* bits of security strength. The *returned_bits* will have full entropy, as stated in Sections 4.2 and 5.2. The *additional_input* is an optional input parameter in the **NRBG_ Generate** call; however, if *additional_input* is provided in the call, it **shall** be included as *additional_input* in the **Generate_function** call.

If the request is not successful (i.e., there is an error), the **NRBG_Generate** function is aborted, and the *status* received in step 3.1 and a Null string are returned to the consuming application (see step 3.2). The handling of status codes by the consuming application is discussed in Section 7.

However, if *status* indicates a success, *returned_bits* contains *s*/2 bits with full entropy.

In steps 3.3 and 3.4, the bitstring returned from step 3.1 (i.e., *returned_bits*) is concatenated with any previously obtained bits, and the amount of entropy received in the returned bits (i.e., *s*/2) is added into the counter. If the total number of full-entropy bits requested by the consuming application has not been obtained yet (i.e., *n* bits), then step 3 continues at step 3.1. Otherwise, the exact number of bits are selected from the collected bitstring and returned to the consuming application (see step 4).

Note that the **Generate_function** call for prediction resistance in step 3.1 requires a call to the DRBG's reseed function, which uses a **Get_entropy_input** call to access the entropy source; a **Get_entropy_input** construction in Section 10.3.3 **shall** be used by the DRBG's **Reseed_function**.

9.4.3 Direct DRBG Access

The DRBG mechanism used by the Oversampling-NRBG may be directly accessed as a normal DRBG using the same or a different instantiation than that used when the DRBG mechanism is performing as part of the NRBG. If the directly accessed DRBG instantiation is the same as the instantiation used for the Oversampling-NRBG construction, then the **DRBG_function** as specified in Section 7.2 is used, and prediction resistance **shall** be performed on every call to the DRBG mechanism. Note that in this case, entropy-source requests are made only once per consuming-application request, rather than for every *s*/2 bits requested by the consuming application, where *s* is the instantiated security strength of the DRBG instantiation used by the NRBG.

If a separate instantiation is used for direct access to the DRBG, then the **Generate_function** as specified in Section 7.2 is used, but a request for prediction resistance is optional. The randomness source for direct DRBG access may be any of those discussed in Section 6, including the entropy source of the Oversampling-NRBG construction. The DRBG **shall** be designed as discussed in Section 8.

When reseeding is required during the generation request (i.e., because prediction resistance is requested or the DRBG instantiation has reached the end of its reseed interval), the **Reseed_function** specified in Section 7.2 and SP 800-90A **shall** be used.

10 Additional Constructions

Additional constructions are required to complete an RBG.The first three sections are used by a target DRBG to access a randomness source.

- Section 10.1 contains constructions to be used to access a source DRBG,
- Section 10.2 contains a construction for accessing an NRBG, and
- Section 10.3 contains constructions to directly access one or more entropy sources.

These constructions include **Get_entropy_input** calls that serve as interfaces between the target DRBG and its randomness source. Figure 4 in Section 8 depicts the use of a randomness source by a target DRBG. The target DRBG invokes a **Get_entropy_input** call, which is, in effect, translated to the appropriate call for the selected randomness source by the interface routines.

Note that when the randomness source of a target DRBG is a chain of RBG's, an appropriate **Get_entropy_input** construction in this section needs to be used by each RBG in the chain to access its randomness source. When the source DRBG for a target DRBG is accessing its own randomness source, this source DRBG becomes a target DRBG during that process. For example, suppose that target DRBG A uses DRBG B as its randomness source, and DRBG B uses DRBG C as its randomness source. When DRBG A uses DRBG B as its randomness source, DRBG A is the target DRBG, and DRBG B is the source DRBG. However, when DRBG B uses DRBG C as its randomness source, DRBG B becomes the target DRBG, and DRBG C is DRBG B's source DRBG.

Section 10.4 provides a construction that will allow a consuming application to obtain full-entropy output directly from a DRBG that supports prediction resistance.

10.1 Constructions for Using a DRBG as a Randomness Source

A target DRBG can use another **approved** DRBG as a randomness source. The source DRBG **shall** generate at least the minimum number of bits and the amount of entropy required to fulfill the **Get_entropy_input** request from the requesting DRBG (i.e., the target DRBG) or return an error indication. When a nonce is required for instantiating the target DRBG, and the nonce is not provided by the application or environment, the source DRBG **shall** also be used to obtain the nonce.

Sections 10.1.1 and 10.1.2 provide constructions for use by a target DRBG to access a source DRBG. The source DRBG **shall not** be the same instantiation as the target DRBG, i.e., the source DRBG may be a completely different DRBG design than the target DRBG, or the same DRBG design but a different instantiation.

This Recommendation assumes that the state handle for the source DRBG is known by the target DRBG (e.g., because of a contractual relationship). Whether or not a source DRBG can provide prediction resistance may also be known, or can be determined by requesting a prediction-resistance capability during instantiation using that DRBG.

Section 10.1.1 specifies a construction that can be used when the security strength to be requested by a target DRBG does not exceed the security strength of the source DRBG. Section 10.1.2 specifies a construction that can be used when a target DRBG could request full entropy or an amount of entropy greater than the security strength of the source DRBG, and the source is known to provide prediction resistance (i.e., the source has access to a Live Entropy Source).

10.1.1 The Requested Security Strength Does Not Exceed the Strength of the Source DRBG

The use of this construction is appropriate when the source DRBG is instantiated at a security strength that is known to be equal to or greater than the security strength to be requested by the target DRBG (e.g., because of a contractual relationship, or because the target DRBG will only request the lowest security strength - 112 bits). The source DRBG may or may not support prediction resistance. Note that when prediction resistance is requested, the source DRBG is reseeded once before providing the requested number of bits to the target DRBG, as opposed to possibly multiple times as may be the case for the construction in Section 10.1.2.

The **Get_entropy_input** call in the target DRBG accesses the source DRBG using the following construction:

Get_entropy_input:

Input: integer (*min_entropy*, *min_length*, *max_length*, *prediction_resistance_request*).

Output: integer *status*, bitstring *returned_bits*.

Process:

1. If (*min_entropy* > *min_length*), then *min_length* = *min_entropy*.
2. If (*min_length* > *max_length*), then return (FAILURE, *Null*).
3. (*status*, *returned_bits*) = **Generate_function** (*state_handle*, *min_length*, *min_entropy*, *prediction_resistance_request*).
4. If (*status* ≠ SUCCESS), then return (*status*, *Null*).
5. If ((**length_in_bits**(*returned_bits*) > *max_length*)), then *returned_bits* = **df**(*returned_bits*, *max_length*).
6. Return (SUCCESS, *returned_bits*).

Steps 1 and 2 check the input parameters and either adjust them (step 1), or return an indication of a failure because the received values are unacceptable, along with a *Null* sring as the *returned_bits* (step 2).

In step 3, the **Generate_function** (see Section 7.2) passes the number of bits to be returned (*min_length*), the minimum security strength that needs to be provided (*min_entropy*) and any prediction-resistance request parameters provided in the **Get_entropy_input** call to the source DRBG indicated by the *state_handle*. Either a *status* code indicating success and the requested bits are returned, or an indication of an error is returned.

The *status* is checked in step 4, and the **Get_entropy_input** routine is aborted if an indication of success was not returned from step 3; in this case, the *status* is returned, along with a Null string as the *returned_bits*. The handling of status codes by the consuming application is discussed in Section 7.

If the length of the *returned_bits* exceeds the maximum length of the bitstring that can be handled (*max_length*), the bitstring is passed through a derivation function from SP 800-90A to compress the bitstring to *max_length* bits (step 5).

In step 6, a *status* code indicating success and the *returned_bits* are returned.

Note that if prediction resistance is requested, the source DRBG will use a reseed function with its own **Get_entropy_input** call; see Section 8.4 for its form.

10.1.2 Accessing a Source DRBG with Prediction Resistance to Obtain any Security Strength

The use of this construction is appropriate when the source DRBG is known to have access to a Live Entropy Source. The source DRBG may be instantiated at any security strength, including a security strength that is less than that of the target DRBG. Multiple calls requesting prediction resistance are made in the **Get_entropy_input** routine of the target DRBG (see below) until a bitstring with sufficient entropy is assembled. The resulting bitstring will have full entropy.

For this construction, either the security strength *s* of the source DRBG **shall** be known (e.g., because of a contractual relationship), or *s* **shall** be set in the request to the minimum security strength of a DRBG in this Recommendation (i.e., *s* = 112).

The following **Get_entropy_input** call can be used to obtain the required amount of entropy:

Get_entropy_input:

Input: integer (*min_entropy*, *min_length*, *max_length*, *prediction_resistance_request*).

Output: integer *status*, bitstring *collected_bits*.

Process:

1. If (*min_entropy* > *min_length*), then *min_length* = *min_entropy*.
2. If (*min_entropy* > *max_length*), then return (FAILURE, *Null*).
3. *collected_bits* = *Null*.
4. *collected_entropy* = 0.
5. While (*collected_entropy* < *min_entropy*)
    - 5.1 (*status*, *tmp*) = **Generate_function** (*state_handle*, *s*/2, *s*, *prediction_resistance_request* = TRUE).
    - 5.2 If (*status* ≠ SUCCESS), then return (*status*, *Null*).
    - 5.3 *collected_bits* = *collected_bits* || *tmp*.
    - 5.4 *collected_entropy* = *collected_entropy* + *s*/2.
6. If ((**length_in_bits**(*collected_bits*) > *max_length*)), then *collected_bits* = **df**(*collected_bits*, *max_length*).
7. Return (SUCCESS, *collected_bits*).

Steps 1 and 2 check the input parameters and either adjust them (step 1), or return an indication of a failure because the received values are unacceptable, along with a *Null* sring as the *returned_bits* (step 2).

The bitstring intended to collect generated bits (*collected_bits*) for return to the calling routine is initialized to the null bitstring in step 3, and a counter for recording the amount of entropy obtained (*collected_entropy*)is initialized to zero in step 4.

Step 5 collects bits generated by the source DRBG indicated by the *state_handle*. Step 5.1 requests that *s*/2 bits be generated by the source DRBG at a security strength of *s* bits; note that even if prediction resistance is not explicitly requested in the **Get_entropy_input** call, the **Generate_function** call requests prediction resistance. If this call is successful, full-entropy bits are returned in *tmp*.

Step 5.2 checks the *status* returned for step 5.1; if the *status* does not indicate a success, then the **Get_entropy_input** routine is aborted; the *status* code is returned, along with a null string as the *returned_bits*. Step 5.3 concatenates the newly acquired bits to any previously obtained bits, and step 5.4 adds in the entropy of the newly acquired bits to the entropy counter. Step 5 is repeated until sufficient entropy has been obtained.

In step 6, if the length of the concatenated bitstring exceeds the maximum length of the bitstring that can be handled (*max_length*), the bitstring is passed through a derivation function from SP 800-90A to compress the bitstring to *max_length* bits.

In step 7, a successful *status* code is returned to the calling application, along with the *collected_bits*.

Note that the source DRBG requires a reseed function with its own **Get_entropy_input** call; see Section 8.4 for its form.

10.2 Construction for Using an NRBG as a Randomness Source

This section specifies a construction for a target DRBG to access an NRBG as the randomness source. An NRBG includes a Live Entropy Source and provides full entropy output. The target DRBG's **Get_entropy_input** call to a source NRBG is fulfilled as follows:

Get_entropy_input:

Input: integer (*min_length*).

Output: integer *status*, bitstring *returned_bits*.

Process:

1. (*status*, *returned_bits*) = **NRBG_Generate**(*state_handle*, *min_length*).
2. If (*status* ≠ SUCCESS), then return (*status*, *Null*).
3. Return (SUCCESS, *returned_bits*).

In step 1, the **NRBG_Generate** function specified in Section 7.3 is called to obtain *min_length* bits. The *state handle* refers to the DRBG instantiation used by the NRBG.

Step 2 checks the status returned for step 1; if the status indicates that the request was not successful, then the **Get_entropy_input** is aborted; the status code is returned, along with a null string as the *returned_bits*.

Otherwise, a successful status code is returned to the calling application, along with the newly generated bits (step 3).

10.3 Constructions for Using an Entropy Source as a Randomness Sources

A single entropy source or multiple entropy sources may be used as a randomness source(s) by a DRBG, and the output of these entropy sources may be externally conditioned before use. Section 10.3.1 discusses the **Get_Entropy** call to be used by an implementation to access entropy sources, including methods for compressing entropy-source output when the entropy rate of the entropy source(s) is very low, and the entropy bits need to be condensed into a shorter bitstring before use. Section 10.3.2 provides guidance for the external conditioning of entropy-source output(s) obtained by the **Get_entropy_input** function prior to use by a DRBG. Section 10.3.3 provides the **Get_entropy_input** constructions to be used by a target DRBG to access one or more entropy sources using a **Get_Entropy** call.

10.3.1 The Get_Entropy Call

The **Get_Entropy** call (used by the **Get_entropy_input** construction in Section 10.3.3) is used to obtain entropy from one or more independent entropy sources. The form of the call is specified in Section 7.4, i.e.,

(*status*, *entropy_bitstring*) = **Get_Entropy**(*requested_entropy*, *max_length*),

where *max_length* is an optional parameter that indicates the maximum length allowed for *entropy_bitstring*. The implementation of this function depends on the entropy sources to be accessed.

The expected behavior of the **Get_Entropy** function is as follows:

1. When a non-null *entropy_bitstring* is returned from a **Get_Entropy** call, the *entropy_bitstring* **shall** contain sufficient entropy to fulfill the request, and the length of the bitstring **shall not** exceed the value of *max_length* (if optionally provided). The *status* **shall** indicate a SUCCESS when and only when these conditions are met.

2. If an error is detected during the execution of the **Get_Entropy** function or sufficient entropy is not currently available, then the **Get_Entropy** function **shall** return a *status* code indicating the problem, along with a null *entropy_bitstring*.

3. The rules for combining the entropy bits produced by one or more entropy sources and determining the assessed entropy are compliant with the assumptions discussed in items 3 and 4 of Section 4.2.

4. When the entropy produced by the entropy source(s) is very long (e.g., because the entropy rate of the entropy source(s) is very low), and the entropy bits may need to be condensed into a shorter bitstring, the **Get_Entropy** function in Section 10.3.1.1 or Section 10.3.1.2 **shall** be used to condense the entropy bits without losing the available entropy in the bit string.

5. If the returned entropy exceeds the requested entropy, *entropy_bitstring* **shall** only be credited with the requested amount of entropy.

6. The **Get_Entropy** function could return a *status* code indicating that entropy is not currently available (e.g., the entropy source(s) returned this indication, or the **Get_Entropy** function has waited for a response from the entropy source(s) for an unacceptable amount of time). In this case, the **Get_entropy** function **shall** return a null *entropy_bitstring*.

Note that in some cases, a short delay could occur before a response is received from the **Get_Entropy** call.

Sections 10.3.1.1 and 10.3.1.2 provide methods for condensing bitstrings containing entropy, when required, during a **Get_Entropy** call. Each of the methods includes a step for querying all available entropy sources. If all available entropy sources indicate fatal errors, than the **Get_Entropy** function **shall** return an error indication and a null value for the *entropy_bitstring* to the routine that called the **Get_Entropy** function (i.e., a **Get_entropy_input** construction provided in Section 10.3.3). If multiple entropy sources are used during the execution of the **Get_Entropy** function, queries may be made to any combination of those entropy sources. Note that if no entropy could be collected from any of the entropy sources, an error indication is returned as the *statu*s code, and a Null bitstring is returned as the *entropy_bitstring* to the routine that called the **Get_Entropy** function.

10.3.1.1 Condensing Entropy Bits during Entropy Collection

The entropy in a bitstring can be condensed during the collection process (e.g., after each access of one or more entropy source(s) using a nonce and derivation function specified in SP 800-90A. The following pseudocode describes the process for the **Get_Entropy** call:

Get_Entropy:

Input: integer (*requested_entropy*, *max_length*).

Output: integer *status*, bitstring *entropy_bitstring*.

Process:

1. If *requested_entropy* > *max_length*, return an error indication and a null value for the *entropy_bitstring.*
2. $n = 2 \times requested\_entropy$.
3. $entropy\_bitstring = 0^n$.
4. *collected_entropy* = 0.
5. While *collected_entropy* < *requested_entropy*

    5.1 Query one or more entropy sources to obtain *queried_bits* and the *assessed_entropy* for those bits. Note that *queried_bits* is the concatenated output of the queried entropy sources, and *assessed_entropy* is the total entropy obtained from those entopy sources. If all available entropy sources indicate fatal errors, then the **Get_Entropy** function returns an error indication and a null value for the *entropy_bitstring.* The requirements for this process are provided in Section 10.3.1.

5.2 *nonce* = **MakeNextNonce**().

5.3 *entropy_bitstring* = *entropy_bitstring* ⊕ **df**((*nonce* || *queried_ bits*), *n*).

5.4 *collected_entropy* = *collected_entropy* + *assessed_entropy*.

6. If (*n* > *max_length*), then *entropy_bitstring* = **df**(*entropy_bitstring*, *max_length*).

7. Return (SUCCESS, *entropy_bitstring*).

Step 1 checks that the requested entropy is not greater than then maximum length of the string to be returned as *entropy_bitstring*.

Step 2 sets the length of the bit string that will be collected using this process; there may be no relationship between the value of *n* and the *max_length* parameter that could optionally be provided in the **Get_Entropy** call. Step 3 initializes the *entropy_bitstring* into which the entropy will be accumulated to all zeros, and step 4 sets the entropy-collection counter to zero.

Step 5 collects the entropy. In step 5.1, one or more entropy sources are queried.

In step 5.2, a *nonce* is determined. The *nonce* **should not** repeat during the lifetime of the target DRBG (i.e., a DRBG instantiation). The target DRBG **shall not** be used to provide this nonce, since there is a (very small) probability that values could repeat. The simplest implementation of **MakeNextNonce** produces a large counter value.

In step 5.3, the *nonce* is combined with the queried bits returned in step 5.1 using a derivation function specified in SP 800-90A, and the *assessed_entropy* from the current query is added into the entropy counter in step 5.4.

After all requested entropy bits are obtained, step 6 checks that the length of the accumulated bitstring does not exceed the *max_length* value that may have been provided as an input to the **Get_Entropy** function, and condenses the entropy_bitstring, if necessary. Note that if *max_length* was not provided, this step is not needed.

In step 7, the collected *entropy_bitstring* is returned to the calling routine (i.e., a **Get_entropy_input** function), along with a status of SUCCESS.

10.3.1.2 Condensing After Entropy Collection

The entropy in a bitstring can be condensed after the entire amount of requested entropy has been collected by the **Get_Entropy** function using a derivation function specified in SP 800-90A. The following pseudocode describes the process for the **Get_Entropy** call:

Get_Entropy:

Input: integer (*requested_entropy*, *max_length*).

Output: integer *status*, bitstring *entropy_bitstring*.

Process:

1. If *requested_entropy* > *max_length*, return an error indication and a null value for the *entropy_bitstring.*
2. *collected_entropy* = 0.
3. *entropy_bitstring* = the Null string.

4. While *collected_entropy* < *requested_entropy*

    4.1 Query one or more entropy sources to obtain *queried_bits* and the *assessed_entropy* for those bits. Note that *queried_bits* is the concatenated output of the queried entropy sources, and *assessed_entropy* is the total entropy obtained from those entopy soucces. If all available entropy sources indicate fatal errors, than the **Get_Entropy** function would return an error indication and a null value for the *entropy_bitstring* to the **Get_Entropy** calling routine (i.e., a **Get_entropy_input** function); the requirements for this process are provided in Section 10.3.1.

    4.2 *entropy_bitstring* = *entropy_bitstring* || *queried_bits*.

    4.3 *collected_entropy* = *collected_entropy* + *assessed_entropy*.

5. *n* = **length_in_bits**(*entropy_bitstring*).

6. If (*n* > *max_length*), then *entropy_bitstring* = **df**(*entropy_bitstring*, *max_length*).

7. Return (SUCCESS, *entropy_bitstring*).

Step 1 checks that the requested entropy is not greater than then maximum length of the string to be returned as *entropy_bitstring*.

Steps 2 and 3 initialize the entropy-collection counter to zero and initialize the bitstring into which the entropy bits will be accumulated to the null sring.

Step 4 collects the entropy. In step 4.1, one or more entropy sources are queried. In step 4.2, the string of *queried_bits* is concatenated to any previously collected bits, and the entropy-collection counter is incremented by the amount of entropy present in the latest collected bits. Step 4 is iterated until sufficient entropy has been collected to fulfill the amount of entropy requested for the **Get_Entropy** call.

After all requested entropy has been obtained, step 5 determines the length of the collected bitstring, and step 6 checks that this length does not exceed the value of *max_length* that may optionally have been provided in the **Get_Entropy** call. Note that if *max_length* was not provided, this step is not needed.

In step 7, the collected *entropy_bitstring* is returned to the calling routine (i.e., a **Get_entropy_input** function), along with a status of SUCCESS.

10.3.2 External Conditioning Functions

Conditioning may be performed on the output of an entropy source prior to use by an RBG (referred to as external conditioning). A conditioning function may be used to distribute the entropy in a bitstring across the entire output of the conditioning function, to condense the entropy in the input bitstring into a shorter bitstring, and can be used to provide a bit string with full entropy.

The external conditioning of entropy-source output is optional within an RBG unless the entropy-source output is used by the XOR-NRBG, and the entropy source does not provide full-entropy output itself (see Figure 7). In this case, external conditioning is required to provide bits with full entropy on the left side of the "⊕" in Figure 7; if the same entropy source is used to seed or reseed the DRBG of the XOR-NRBG, external conditioning is not required.

When external conditioning is performed, a vetted or referenced conditioning function from [SP 800-90B] **shall** be used.

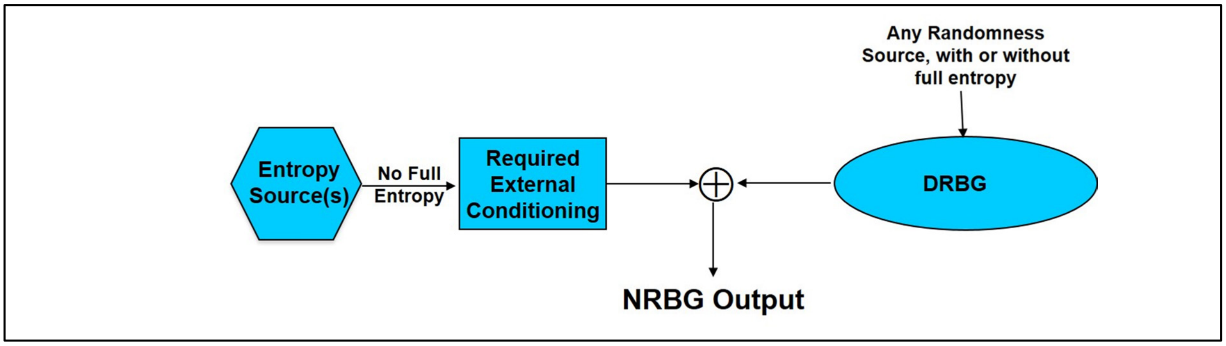

Figure 7: XOR-NRBG Requiring External Conditioning

10.3.2.1 Using an External Conditioning Function

Figure 8 depicts the process of collecting entropy from one or more entropy sources and conditioning the resulting *entropy_bitstring*, which is a concatenation of the output of the entropy source(s).

When (optional) external conditioning is performed, one of the vetted conditioning functions listed or referenced in [SP 800-90B] **shall** be used. A conditioning function **shall** be selected such that the maximum amount of entropy to be requested using a **Get_entropy_input** call is no greater than the length of the conditioning function's output block, i.e.,

$$min\_entropy \leq n_{out},$$

where,

For a hash function, HMAC, and Hash_df:

n_{out} = the length of the hash function output block.

For CMAC and CBC-MAC:

n_{out} = the length of an AES block (128 bits).

For Block_Cipher_df:

n_{out} = the length of the AES key (128, 192 or 256 bits).

10.3.2.2 Keys Used for External Conditioning

For the keyed external conditioning functions (e.g., HMAC, CMAC and CBC-MAC), the key

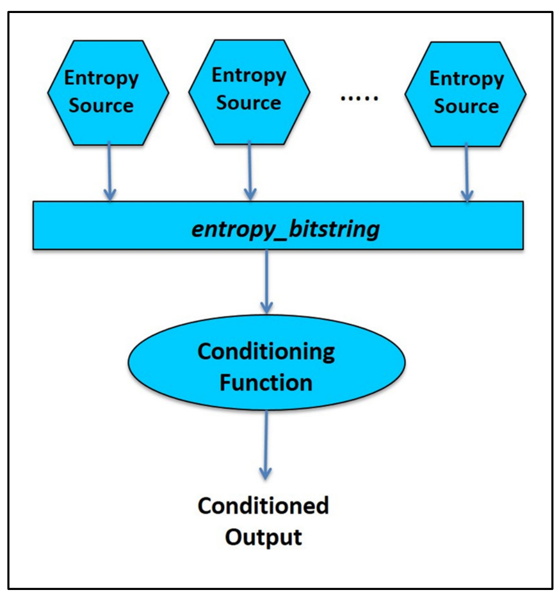

Figure 8: Using a Conditioning Function

should be generated randomly each time that an RBG powers up. The key could be obtained by using entropy bits from the entropy source(s) with at least *m* bits of assessed entropy and a minimum length of *keylen* bits, where *m* is the security strength to be provided in the key, and *keylen* is the length of the key, i.e.,

HMAC: $m \geq \{112, 128, 192, 256\}$; $keylen = m$.

AES-128: $m = keylen = 128$.

AES-192: $m = keylen = 192$.

AES-256: $m = keylen = 256$.

When the length of the acquired *entropy_bitstring* is greater than *keylen* bits, the entropy bit string needs to be compressed to the appropriate key length using a derivation function from SP 800-90A to determine the key:

$$Key = \mathbf{df}(entropy\_bitstring, keylen),$$

where **df** is either **Hash_df** or **Block_Cipher_df**.

When **Hash_df** is used for compressing the entropy bits, the preimage security strength of the hash function used in the derivation function **shall** meet or exceed the value of *m.*

When **Block_Cipher_df** is used for compressing the entropy bits, the key used by the derivation function itself may be an arbitrary value.

10.3.3 Get_entropy_input Constructions for Accessing Entropy Sources

Section 10.3.3.1 provides a **Get_entropy_input** construction for the case where a conditioning function is not used. Section 10.3.3.2 provides a construction for obtaining entropy and using a conditioning function to compress entropy into a shorter bitstring when full entropy output is not required. Section 10.3.3.3 provides a construction for obtaining full entropy using a conditioning function.

10.3.3.1 Construction When a Conditioning Function is not Used

This construction is appropriate when the RBG can use the entropy-source output as produced, except for any condensing of the entropy bitstring as specified in Section 10.3.1.1 or 10.3.1.2. If full-entropy output is required from this construction, an entropy source **shall** have been selected that provides it without further processing.

In this construction, the target DRBG makes a **Get_entropy_input** call to obtain entropy bits from the entropy source(s), indicating the min-entropy required. The **Get_entropy_input** function below accesses the entropy source(s) using the **Get_Entropy** call discussed in Section 10.3.1. An explicit request for prediction resistance in the **Get_entropy_input** request is not required, since the entropy source(s) are already being invoked in the construction.

Get_entropy_input:

Input: integer (*min_entropy*, *max_length*).

Output: integer *status*, bitstring *entropy_bitstring*.

Process:

1. (*status, entropy_bitstring*) = **Get_Entropy**(*min_entropy, max_length*).

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

3. Return SUCCESS, *entropy_bitstring*.

In step 1, the entropy bits are requested from the entropy source using a **Get_Entropy** function; the specifics of this function depend on the entropy source(s) to be used. The returned *status* from step 1 is checked in step 2; if the *status* indicates that the call was not successful, the received *status* and a *Null* string are returned from the **Get_entropy_input** function.

In step 3, an indication of SUCCESS and the *entropy_bitstring* are returned.

10.3.3.2 Construction When a Vetted Conditioning Function is Used and Full Entropy is Not Required)

When an external conditioning function is used to process entropy-source output, any of the vetted conditioning functions listed or referenced in SP 800-90B may be used, providing that the entropy requested by the DRBG mechanism is no greater than the length of the conditioning function output (n_{out}), as specified in Section 10.3.2.1.

The following construction will compress the entropy contained in the input string into a string of n_{out} bits. The entropy in the output string will be distributed uniformly across the output string; therefore, the entire output string **shall** be used as entropy input for the DRBG.

Get_entropy_input:

Input: integer (*min_entropy*).

Output: integer *status*, bitstring *entropy_bitstring*.

Process:

1. If (*min_entropy* > n_{out}) then return(*status*, *Null*), where *status* indicates an error condition.
2. (*status, entropy_bitstring*) = **Get_Entropy**(*min_entropy*).
3. If (*status* ≠ SUCCESS), then return (*status*, *Null*).
4. *output_bitstring* = **Conditioning_function**(*entropy_bitstring*).
5. Return (SUCCESS, *output_bitstring.*

Step 1 checks that the amount of entropy requested can be handled by the conditioning function, returning an error indication as the *status* and a *Null* string.

Step 2 requests the entropy from the entropy source(s), and step 3 checks whether or not there was an error returned as the *status* in step 2. If *status* indicated an error, the *status* and a *Null* string are returned to the calling routine. Note that the **Get_Entropy** call does not require a *max_length* parameter, since the **Conditioning_function** in step 4 will condense the entropy_bitstring to n_{out} bits.

Step 4 invokes the conditioning function for processing the *entropy_bitstring* obtained from step 2. The specific **Conditioning_function** call is specified in Section 7.5.

Step 5 returns the conditioned result.

10.3.3.3 Construction When a Vetted Conditioning Function is Used to Obtain Full Entropy Bitstrings

This construction will produce full-entropy bits as output (e.g., for the XOR-NRBG when the entropy source does not provide full-entropy output). Any of the vetted conditioning functions listed or referenced in SP 800-90B may be used, providing that the entropy requested by the DRBG mechanism is no greater than the length of the conditioning function output (n_{out}), as specified in Section 10.3.2.1.

In the construction below, the target DRBG makes a **Get_entropy_input** call to obtain entropy from one or more entropy sources, indicating the min-entropy required; any condensing of the entropy source output into shorter bitstrings **shall** have been performed using one of the methods in Section 10.3.1.

Get_entropy_input:

Input: integer (*min_entropy*).

Output: integer *status*, bitstring *entropy_bitstring*.

Process:

1. If *(min_entropy* > n_{out}) then return(*status*, *Null*), where *status* indicates an error condition.
2. (*status*, *entropy_bitstring*) = **Get_Entropy**(2 × n_{out}).
3. If (*status* ≠ SUCCESS), then return (*status*, *Null*).
4. (*status*, *returned_bitstring*) = **Conditioning_function**(*entropy_bitstring*).
5. *entropy_bitstring* = **leftmost**(*entropy_bitstring*, *min_entropy*).
6. Return SUCCESS, *entropy_bitstring*.

Step 1 checks that the amount of entropy requested can be handled by the conditioning function, returning an error indication as the *status* and a *Null* string.

Step 2 requests an amount of entropy from the entropy source(s) that is twice the length of the conditioning-function outut block, and step 3 checks whether or not there was an error returned as the *status* in step 2. If *status* indicated an error, the *status* and a *Null* string are returned to the calling routine. Note that the **Get_Entropy** call does not require a *max_length* parameter, since the **Conditioning_function** in step 4 will condense the entropy_bitstring to n_{out} bits.

Step 4 invokes the conditioning function for processing the *entropy_bitstring* obtained from step 2. The specific **Conditioning_function** call is specified in Section 7.5.

Step 5 truncates the conditioning function output to the number of bits requested in the **Get_entropy_input** call, and step 6 returns the result.

10.4 General Construction Using a DRBG with Prediction Resistance to Obtain Full-Entropy Output Upon Request

A DRBG with a Live Entropy Source that provides prediction resistance can also be used to provide full-entropy output when requested. The following construction can be used by a consuming application to request bits from a DRBG with and without prediction resistance, and with and without requesting full-entropy output. The construction is divided into two paths; the path used depends on whether full-entropy output is requested.

When full entropy is not requested, the DRBG is requested to generate bits normally, i.e., without special processing.

When full entropy is requested, multiple calls are made to the DRBG to obtain the number of bits and the entropy needed by the consuming application. Each call requests that *s*/2 bits be returned, where *s* is the security strength requested. The value of *s* requested depends on the randomness source, and it is up to the developer to select an appropriate value. If the randomness source is known to be an entropy source, then any of the **approved** security strengths can be requested. If the randomness source is known to be a source DRBG, and the security strength supported by that source DRBG is known, then *s* can be a value that does not exceed the source DRBG's security strength; otherwise, the use of the lowest security strength supported by this Recommendation is recommended (i.e., 112 bits).

Let *s* be an appropriate security strength for the randomness source to be used.

General_DRBG_Generate:

Input: integer (*state_handle*, *requested_number_of_bits*, *security_strength*, *full_entropy_request*, *prediction_resistance_request*), string *additional_input*.

Output: integer *status*, bitstring *returned_bits*.

Process:

1. If (*full_entropy_request* = TRUE), then

    Comment: Full entropy has been requested.

    1.1 *returned_bits* =*Null*.

    1.2 *sum* = *0*.

    1.3 While (*sum* < *requested_number_of_bits*)

        1.3.1 (*status*, *tmp*) = **Generate_function**(*state_handle*, *s*/2, *s*, *prediction_resistance_request* = TRUE, *additional_input*).

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

        1.3.3 *returned_bits* = *returned_bits* || *tmp*.

        1.3.4 *sum* = *sum* + *s*/2.

        Comment: Use a null string as the *additional_input* for subsequent iterations of the While loop.

        1.3.5 *additional_input* = *Null*.

    1.4 Return SUCCESS and **leftmost**(*returned_bits*, *requested_number_of_bits*).

    Comment: Full entropy output has not been requested.

2. (*status*, *returned_bits*) = **Generate_function**(*state_handle*, *requested_number_of_bits, security_strength*, *prediction_resistance_request*, *additional_input*).

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

4. Return (SUCCESS, *returned_bits*).

Step 1 handles the case in which full entropy is requested.

- A bitstring intended to collect entropy bits for return to the calling routine (i.e., *returned_bits*) is initialized to the null bitstring in step 1.1, and a counter for recording the amount of entropy obtained (i.e., *sum*) is initialized to zero in step 1.2. Step 1.3 is iterated until the requested number of bits is collected.

- Step 1.3.1 uses a **Generate_function** call to obtain *s*/2 bits with full entropy during each request. The appropriate values from the **General_DRBG_Generate** call are used as input during the **Generate_function** call. The **Generate_function** makes a **Get_entropy_input** request, which is fulfilled using an appropriate construction in Section 10.1, 10.2 or 10.3. Note that the prediction_resistance_request parameter for the **Generate_function** call is set to TRUE so that the DRBG is alerted that the Live Entropy Source must be accessed. Also, note that the *security_strength* and *prediction_resistance_request* input parameters in the **General_DRBG_Generate** request are ignored when full entropy is requested in this path.

- Step 1.3.2 checks whether the *status* returned from step 1.3.1 indicates a SUCCESS; if not, then the *status* code is returned to the consuming application, along with a *Null* string as the *returned_bits*.

- Steps 1.3.3 and 1.3.4 concatenate the bits obtained from step 1.3.1 to any previously acquired bits and adds the amount of entropy obtained into the entropy counter (*sum*). Step 1.3.5 sets any additional input provided in the **General_DRBG_Generate** call to the *Null* string.

- In step 1.4, the requested number of full-entropy bits are returned to the consuming application.

Steps 2-4 handle the case in which the DRBG is requested to provide output, with or without prediction resistance, but not with full entropy.

- Step 2 issues a **Generate_function** call, using the input parameters provided in the **General_DRBG_Generate** call; note that prediction resistance may or may not be requested, in this case.
- Step 3 checks whether the *status* returned from step 2 indicates a SUCCESS; if not, then the *status* code is returned to the consuming application, along with a *Null* string as the *returned_bits*.
- Otherwise, the *returned_bits* provided in step 2 are returned to the consuming application, along with a *status* code of SUCCESS.

11 Combining RBGs

11.1 Discussion

RBGs may be combined if at least one of the RBGs is **approved**. Combining RBGs might be appropriate for a number of reasons, including:

- The desire to use an unapproved DRBG that is believed to be superior in security over an **approved** DRBG,
- The desire to combine DRBGs or NRBGs that use different entropy sources or are based on different components or design principles for increased assurance, or
- The desire to combine RBGs from different implementers or RBGs that are contained in different modules in order to obtain increased assurance.

Combining RBGs is a method of meeting the requirements of this Recommendation, while gaining any security properties provided by other RBGs in which the RBG designer may have confidence. Designs that incorporate DRBGs that are not approved in this Recommendation, but which are believed by the designer to be highly secure, are good candidates for use in a combined RBG.

The construction for combining RBGs provides assurance that the resulting combined RBG will be no weaker than the strongest **approved** component RBG, assuming that the sources of entropy are independent (i.e., different independent entropy sources are used, or the entropy input for a DRBG is used only for that DRBG). Note, however, that there is no assurance that the combined RBG will be substantially stronger than the strongest component RBG.

11.2 Construction to Combine RBGs

11.2.1 Overview

This construction allows *N* component RBGs, at least one of which is **approved**, to be combined to make a new approved RBG.

The requirements, security strength and properties of the combined RBG are as follows:

- The combined RBG construction **shall** include at least one **approved** RBG that is constructed in accordance with this Recommendation. The combined RBG **shall** only be considered to be operating correctly if at least one **approved** RBG in the construction is operating correctly. An **approved** RBG **shall** use an **approved** randomness source; unapproved RBGs may use unapproved randomness sources. However, multiple RBGs **shall not** use the same outputs from a given randomness source.
- The combined RBG has a claimed security strength equal to the highest security strength provided by any **approved** component RBG. Note that if one of the **approved** component RBGs is an NRBG, then the combined RBG can support any security strength when the entropy source of the NRBG is operating correctly. In this case, output from the combined RBG may be used in exactly the same way as the output of any **approved** NRBG. If the entropy source within an **approved** NRBG fails without detection, and no other **approved** NRBG is used within the combined RBG, then the

security strength of the combined RBG is reduced to the security strength of the **approved** DRBG within the combined RBG that has the highest security strength. For example, if a combined RBG consists of an **approved** NRBG and a non-approved DRBG, then if the entropy source within the NRBG fails without detection, the security strength of the combined RBG is reduced to the security strength of the DRBG mechanism within the NRBG.

- The combined RBG is capable of supporting prediction resistance and full entropy requests if either:
  - One of its **approved** component RBGs is an NRBG, or
  - One of its **approved** component RBGs with the same security strength as the combined RBG supports prediction resistance (i.e., a Live Entropy Source is available) and uses the **Get_entropy_input** construction in Section 10.1.2.

The following convention is used to specify a combined RBG: If a component RBG cannot support one or more of the input parameters, those parameters are omitted from the function call. For example, if a given DRBG, *R*, does not support the *requested_security_strength*, *additional_input* and *prediction_resistance_request* parameters in its generate function, then the pseudocode of

(*status*, *returned_bits*) = **Generate_function**(*requested_number_of_bits*, *requested_security_strength*, *prediction_resistance_request*, *additional_input*)

may be substituted by

(*status*, *returned_bits*) = **Generate_function**(*requested_number_of_bits*).

for that DRBG.

Note that all **approved** NRBGs have DRBG mechanisms.

11.2.2 Combined RBG Instantiation

Let $highest\_DRBG\_security\_strength_i$ be the highest possible security strength for R_i, and let N be the number of RBGs in the combined RBG.

Let **MakeNextNonce** be a method for creating a value that is of a fixed-length that **shall not** repeat during the lifetime of the combined RBG. Note that an RBG **shall not** be used to provide this nonce, since there is a (very small) probability that values could repeat.

Instantiation can be summarized by the following:

Combined_Instantiate:

Input: integer (*requested_instantiation_security_strength*, *prediction_resistance_flag*), string *personalization_string.*

Output: string *status*, integer($state\_handle_1$,…$state\_handle_N$).

Process:

1. For i = 1 to N

   1.1 If (R_i supports a personalization string), then

1.1.1 *nonce* = **MakeNextNonce**(). Comment: Use a nonce to create a unique *personalization_string* for each DRBG mechanism that can use it.

1.1.2 *modified_personalization_string* = *nonce* || *personalization_string*.

Comment: Note that the length of the *modified_personalization_string* **shall not** exceed the maximum allowed length of the personalization string for R_i.

1.2 If R_i is an **approved** NRBG, then

1.2.1 If R_i supports a personalizalization string, then

(*status, state_handle*$_i$) = **NRBG_Instantiate** (*modified_personalization_string*).

Else (*status, state_handle*$_i$) = **NRBG_Instantiate** ().

1.2.2 If *status* indicates an error, then return the *status*, and a *Null* string for each expected *state_handle*.

1.3 If R_i is an **approved** DRBG, then

1.3.1 If R_i supports a personalization string, then

(*status, state_handle*$_i$) = **Instantiate_function**(*requested_instantiation_security_strength*, *prediction_resistance_flag*, *modified_personalization_string*).

Else **Instantiate_function**(*requested_instantiation_security_strength*, *prediction_resistance_flag*).

1.3.2 If *status* indicates an error, then return *status* and a *Null* string as the *state_handle* for each expected *state_handle.*

Note: Instantiate the DRBG mechanism with the parameters that are provided in the **Combined_Instantiate** call that are supported for the instantiation of the DRBG. The *prediction_resistance_request_flag* **shall** be present in step 1.3.1 and set to TRUE if prediction resistance will be requested in the **Generate_function** request.

1.4. If R_i is ***not*** an **approved** RBG, and R_i contains a DRBG mechanism, then

1.4.1 Instantiate the unapproved DRBG(s) with any implemented parameters that are provided in the **Combined_Instantiate** call that are supported by the DRBG. If a *personalization_string* can be used, let the personalization string provided to the DRBG be *modified_personalization_string*. Set *state_handle*$_i$ equal to the returned state handle, if appropriate; otherwise, set *state_handle*$_i$ equal to a value that indicates that there is no state handle.

1.4.2 If an error is indicated, return the error indicator as the *status*, and a *Null* string for each expected *state_handle*.

Else:

1.4.3 Instantiate the unapproved DRBG with any implemented parameters that are provided in the **Combined_Instantiate** call that are supported. Obtain a *state_handle*, if appropriate.

1.4.4 If an error is indicated, return an error indicator as the *status*, and a *Null* string for each expected *state_handle*.

2. Return SUCCESS and any state handles.

Note that if an unapproved RBG does not have a DRBG mechanism, instantiation is not performed for that RBG.

The *prediction_resistance_flag* and *personalization_string* input parameters are optional in the **Combined_Instantiate** call; however, if either one or both are provided, they **shall** be passed to any component RBG that supports their use.

The following requirement applies to the instantiation of DRBG mechanisms in this construction:

- Each component DRBG **shall** be provided with a different bitstring containing entropy; the bitstrings may be obtained from the same or different randomness sources, but multiple component DRBGs **shall not** use any portion of the same bitstring (e.g., if the randomness source provides a very long bitstring from which multiple DRBG are assigned subsets of bits for instantiation, then the subsets **shall** be disjoint). The length of the bitstring used by each DRBG **shall** be less than or equal to the maximum length allowed for that DRBG mechanism and **shall** contain sufficient entropy for the DRBG's security strength.

11.2.3 Combined RBG Reseeding

Each DRBG mechanism component of an RBG may be reseeded independently at any time, and may control its own reseeding. However, if the consuming application requests a reseed, this **shall** be performed on all component DRBG mechanisms capable of being reseeded as follows:

Combined_Reseed:

Input: integer($state\_handle_1$, …, $state\_handle_N$, *prediction_resistance_request*), string *additional_input.*

Output: string *status*.

Process:

1. For *i* = 1 to *N*

   1.1 If R_i *is* an **approved** NRBG

      1.1.1 *status* = **NRBG_Reseed**($state\_handle_i$, *additional_input*).

      1.1.2. If *status* indicates an error, then return (*status*).

1.2 If R_i *is* an **approved** DRBG

1.2.1 *status* = **Reseed_function**(*state_handle*$_i$, *prediction_resistance_request*, *additional_input*).

1.2.2. If *status* indicates an error, then return (*status*).

Note: Reseed the DRBG mechanism with prediction resistance and *additional_input* if these parameters are supported.

1.3 If R_i is not an **approved** RBG, and R_i contains a DRBG mechanism

1.3.1 Reseed the DRBG with prediction resistance and *additional_input* if these parameters and the reseed function are supported, using the appropriate *state_handle*, if supported.

1.3.2. If an error is indicated, then return the error indicator as the *status*.

2. Return (SUCCESS).

Note that an unapproved RBG that does not contain a DRBG mechanism will not be reseeded.

11.2.4 Combined RBG Generation

The combined RBG generate function is as follows:

Combined_Generate:

Input: integer(*state_handle*$_1$, …, *state_handle*$_N$, *requested_number_of_bits, requested_security_strength, prediction_resistance_request*), string *additional_input.*

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

Process:

1. If prediction resistance is requested, and prediction resistance is not supported by any **approved** RBG within the combined RBG, then return an error indicator as the *status*, and a *Null* string as the *returned_bits*.
2. *tmp* = $0^{requested\_number\_of\_bits}$.
3. For *i* = 1 to *N*

3.1 If R_i is an **approved** NRBG:

3.1.1 (*status*, *returned_bits*) = **NRBG_Generate**(*state_handle*$_i$, *requested_number_of_bits, additional_input*).

3.1.2 If *status* indicates an error, return (*status*, *Null*).

3.2 If R_i is an **approved** DRBG:

3.2.1 (*status, returned_bits*) = **Generate_function**(*state_handle*$_i$, *requested_number_of_bits*, *requested_security_strength*, *prediction_resistance_request*, *additional_input*).

Note: Generate bits using the **approved** DRBG with the parameters provided in the **Combined_Generate** call that are supported.

3.2.2 If *status* indicates an error, return (*status*, *Null*).

3.3 If R_i is not an **approved** RBG:

3.3.1 Generate the requested number of bits using the unapproved DRBG with the parameters provided in the **Combined_Instantiate** call that are supported. Let *status* be the returned status, and *returned_bits* be the returned bits.

3.3.2 If *status* indicates an error, return (*status*, *Null*).

3.4. $tmp = tmp \oplus returned\_bits$.

4. Return SUCCESS, *tmp*.

No intermediate values for *tmp* or outputs of individual RBGs used to generate this combined output **shall** be accessible from outside the boundary or sub-boundary of the combined RBG.

12 Testing

Two types of testing are specified in this Recommendation that may be performed on an RBG: health testing and implementation-validation testing. Health testing **shall** be performed on all RBGs that claim conformance with this Recommendation (see Section 12.1). Section 12.2 provides information on implementation validation.

12.1 Health Testing

Health testing is the testing of an implementation prior to and during normal operation (e.g., periodically) to determine that the implementation continues to perform as expected and as validated. Health testing is performed by the RBG itself, i.e., the tests are designed into the RBG implementation. Two types of tests **shall** be performed: behavior tests and known-answer tests.

- Behavior tests are statistical tests that are performed on the parts of an implementation for which an exact response cannot be predicted. These tests are conducted at startup and continuously thereafter. Such tests are specified in SP 800-90B for noise sources.
- Known-answer tests are performed on the deterministic parts of an implementation (e.g., on an encoded algorithm) and are appropriate for the DRBG mechanisms in SP 800-90A, on the RBG constructions in SP 800-90C, and may be appropriate for deterministic components within SP 800-90B.

The deterministic components of an RBG are normally less likely to fail than the components for which behavior testing is required. Therefore, known-answer tests may be performed less frequently than behavior tests.

An RBG **shall** support the health tests specified in SP 800-90A and SP 800-90B, as well as performing health tests on the components of SP 800-90C and the RBG as a whole. SP 800-90A specifies the use of known-answer tests, and SP 800-90B specifies the use of both behavior and known-answer tests.

The strategy for testing the RBG as a whole is to test the layers of components recursively, using known-answer tests, where appropriate, in order to verify the correct operation of the parts of the RBG that are not simply components from SP 800-90A or SP 800-90B.

12.1.1 Testing RBG Components

Whenever an RBG receives a request to startup, or receives a specific request to perform health testing, a request for health testing **shall** be issued to any DRBG component or randomness-source component within the device receiving the request (e.g., within the sub-boundary receiving the testing request).

When the randomness source consists of a chain of RBGs within a single device:

- If the previous RBGs in the chain are not tested separately, then the health test request **shall** completely test all RBGs in the chain, triggering health tests of all the accessible RBGs that constitute the randomness source[4].

[4] When the RBG boundaries for the chain of RBGs are distributed, it may not be feasible to test all RBGs in

- Any higher-level RBGs in the chain that are tested separately from this test **should** provide an indication of testing success or failure to subsequent RBGs in the chain.
- The entropy source for the target RBG (or the initial RBG in the chain of RBGs) **shall** also be given a health test request as soon as it is available.

The results of the tests **should** propagate down to the target RBG. If any component of the RBG (or chain of RBGs) fails a health test, then the target RBG fails the health test.

12.1.2 Known-Answer Testing for SP 800-90C Components

Known-answer tests **shall** be performed on constructions used by an implementation prior to the first use of the RBG after startup. A known-answer test **shall** be performed on each implemented construction, or on logical sets of constructions. When a construction is grouped with different subsets of other constructions, each such group **shall** be tested. For example, if construction A is used with construction B to execute one process, and with constructions B and C to execute a different process, then all components of each set of constructions **shall** be tested.

12.1.3 Handling Failure

When a failure is detected in an RBG component and reported to the RBG-as-a whole, the RBG **shall** enter an error state. For example, if the entropy source reports that an unrecoverable error has occurred in the noise source, the RBG needs to enter an error state.

SP 800-90A and SP 800-90B discuss the error handling of DRBG mechanisms and entropy sources, respectively. The consuming application for the RBG **shall** be informed when the RBG enters an error state; it is the responsibility of the consuming application to handle the error (e.g., by requesting further guidance from the user or preventing further random bit generation requests).

12.2 Implementation Validation

Implementation validation is the process of verifying that an RBG and its components fulfill the requirements of this Recommendation. An RBG is validated by:

- Validating the components from SP 800-90A and SP 800-90B.
- Validating the use of the constructions in SP 800-90C via code inspection or known-answer tests or both, as appropriate.
- Using known-answer tests to validate the integer/bit conversion routines in SP 800-90A.
- Validating that the appropriate documentation as specified in SP 800-90C has been provided (see below).

Documentation **shall** be developed that will provide assurance to users and testers that an RBG that claims conformance to this Recommendation has been implemented correctly. This documentation **shall** include the following as a minimum:

---

the chain.

- An identification of the construction(s) and components used for the RBG, including a diagram of the interaction of these construction(s) and components.
- Appropriate documentation as specified in SP 800-90A and SP 800-90B; if either the DRBG mechanism or the entropy source has been validated for conformance to SP 800-90A or SP 800-90B, respectively, the appropriate validation certificate **shall** also be provided.
- An identification of the features supported by the RBG (e.g., access to the underlying DRBG mechanism by an NRBG, etc.).
- A description of the health tests performed, including an identification of the periodic intervals for performing the tests.
- A description of any support functions other than health testing.
- A discussion about how the integrity of the health tests will be determined subsequent to implementation validation.
- A discussion about the grouping of constructions for health testing (see Section 12.1.2).
- A description of the RBG components within the RBG boundary.
- If the RBG is distributed, a description about how the RBG is distributed, how each distributed portion is constructed, and the secure channel that is used to transfer information between the sub-boundaries (see Section 5.1).

Appendix A: Diagrams of Basic RBG Configurations

RBGs may be implemented in a variety of ways. Several common configurations are provided as examples below.

A.1 Example Using an XOR Construction

The XOR construction for an NRBG is specified in Section 9.3, and requires a DRBG mechanism and a source of full-entropy bits.

The entropy source itself does not provide full-entropy output, so an external conditioning function is used, say the **Hash_df** specified in SP 800-90A using SHA-1 as the hash function.

The HMAC_DRBG specified in SP 800-90A will be used as the DRBG mechanism, with SHA-1 used as the underlying hash function for the DRBG. The DRBG will obtain its entropy input from the NRBG's entropy source as shown in Figure A-1, i.e., the DRBG uses the NRBG's entropy source as a Live Entropy Source. Bits with full entropy are not required for input to the DRBG, i.e., the output from the entropy source is not externally conditioned before entering the DRBG.

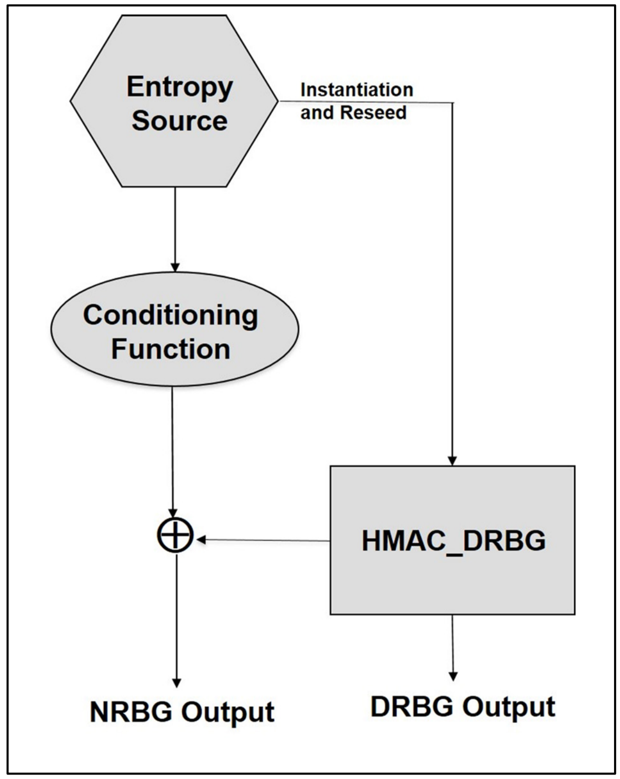

Figure A-1: XOR-NRBG Construction Example

As specified in Section 9.3, the DRBG must be instantiated (and reseeded) at the highest security strength possible for the implemented DRBG mechanism. Since SHA-1 will be used as the underlying hash function of the DRBG, the highest security strength that can be supported by the DRBG mechanism is 128 bits; see SP 800-90A for the **approved** security strengths that are supported for the HMAC_DRBG, and SP 800-57, Part 1 for the security strengths provided by hash functions used for random number generation. Therefore, the DRBG will be instantiated and reseeded at a 128-bit security strength.

Calls are made to the NRBG using the NRBG calls specified in Section 7.3. For this example, all components are contained within a single RBG boundary.

The DRBG mechanism itself can be accessed directly using the same instantiation employed for NRBG calls, using the **NRBG_DRBG_Generate** call specified in Section 7.3. Since the NRBG's Live Entropy Source is always available, the DRBG can support prediction resistance.

If the entropy source produces output at a slow rate, a consuming application might call the NRBG only when full entropy bits are required, obtaining all other output directly from the NRBG's DRBG.

This example provides the following capabilities:

- Full entropy output by the NRBG,

- Fallback to the security strength provided by the DRBG (128 bits) if the entropy source has an undetected failure,
- Direct access to the NRBG's DRBG for faster output,
- DRBG instantiated at a security strength of 128 bits,
- Access to a Live Entropy Source to instantiate and reseed the DRBG, and
- Prediction resistance support for the DRBG when directly accessed, but not during NRBG requests.

A.1.1 NRBG Instantiation

NRBG instantiation includes the instantiation of the DRBG in the XOR construction (see Section 9.3.1). The **NRBG_ Instantiate** construction is:

NRBG_Instantiate:

Input: bitstring *prediction_resistance_flag*, *personalization_string*.

Output: integer *status*.

Process:

Comment: The **Instantiate_function** is specified in SP 800-90A.

1. *status* = **Instantiate_function**(128, *prediction_resistance_flag* = TRUE, *personalization_string*).
2. Return *status*.

Note that in step 1, the *requested_security_strength* parameter has been set to 128 bits, and that a *state_handle* is not returned for this example, since only a single DRBG instantiation will be available. Since prediction resistance will be supported by the DRBG when directly accessed, the *prediction_resistance_flag* is set to TRUE. During the **Instantiate_function** call, a **Get_entropy_input** call will be invoked to obtain entropy bits to instantiate the DRBG mechanism. The **Get_entropy_input** call is fulfilled using the construction in Section 10.3.3.3 using Hash_df and SHA-1.

The **Get_entropy_input** call within the **Instantiate_function** is:

(*status*, *returned_bits*) = **Get_entropy_input**(128, 512).

This call sets the values of *min_entropy* to 128 bits, and *max_length* to 512 bits.

Note that the status returned from the **Instantiate_function** is passed to the consuming application in this example.

A.1.2 NRBG Generation

The NRBG can be called by a consuming application to generate output with full entropy. The construction in Section 9.3.2 is used as follows:

NRBG_Generate:

Input: integer *n*, string *additional_input*.

Output: integer *status*, bitstring *returned_bits*.

Process:

Comment: For step 1, use the construction in Section 10.3.3.3 to obtain and condition the entropy-source output for full entropy.

1. (*status*, *entropy_bitstring*) = **Get_entropy_input**(*n*, *n*).
2. If (*status* ≠ SUCCESS), then return *status*, *Null*.

Comment: For step 3, the **Generate_function** is specified in SP 800-90A.

3. (*status*, *drbg_bits*) = **Generate_function**(*n*, 128, *prediction_resistance_request* = FALSE, *additional_input*).
4. If (*status* ≠ SUCCESS), then return *status*, *Null*.
5. *returned_bits* = *entropy_bitstring* ⊕ *drbg_bits*.
6. Return SUCCESS, *returned_bits*.

Note that the *state_handle* parameter is not used in the **NRBG_Generate** call or the **Generate_function** call (in step 3), since a *state_handle* was not returned from the **NRBG_Instantiate** function (see Appendix A.1.1).

In step 1, the entropy source is accessed using the **Get_entropy_input** routine specified in Section 10.3.3.3 to obtain *n* bits with full entropy.

Step 2 checks that the **Get_entropy-input** call in step 1 was successful; if not, the **NRBG_Generate** function is aborted, returning the received *status* code to the consuming application, along with a *Null* string as the *returned_bits*.

Step 3 calls the DRBG mechanism to generate bits to be XORed with the output of the entropy source in order to produce the NRBG output. Note that a request for prediction resistance is not made in the **Generate_function** call (see Section 9.3.2).

Step 4 performs the same checks as step 2.

In step 5, the *entropy_bitstring* returned in step 1, and the *drbg_bits* obtained in step 3 are XORed together, and the result returned to the consuming application (step 6).

A.1.3 Direct DRBG Generation

The NRBG's DRBG mechanism can be directly accessed by a consuming application using the **NRBG_DRBG_Generate** call specified in Section 7.3. For this example, the **NRBG_DRBG_Generate** function is as follows:

NRBG_DRBG_Generate:

Input: integer (*n*, *security_strength*, *prediction_resistance_request*), bitstring (*additional_input*).

Output: integer *status*, bitstring *returned_bits*.

Process:

1. (*status*, *returned_bits*) = **Generate_function**(*n*, *security_strength*, *prediction_resistance_request*, *additional_input*).

2. Return *status*, *returned_bits*.

Note that the *state_handle* parameter is not used in this example. A request for prediction resistance is optional, and the NRBG's entropy source is the randomness source for any prediction resistance request. The *security_strength* parameter must be less than or equal to 128, for this example.

If prediction resistance is requested, the **Generate_function** calls a **Reseed_function** (see Appendix A.1.4).

A.1.4 DRBG Reseeding

The DRBG must be reseeded at the end of its designed reseed interval, whenever prediction resistance is requested during direct DRBG generate requests (see Appendix A.1.3) and may be reseeded on request (e.g., by the consuming application). Reseeding will be automatic whenever the end of the DRBG's reseed is reached during a **Generate_function** call and when prediction resistance is requested for the **Generate_function** (see the **Generate_function** specification in SP 800-90A). For this example, whether reseeding is done automatically during a **Generate_function** call, or is specifically requested by a consuming application, the **Reseed_function** call is:

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

The **Reseed_function** is specified in SP 800-90A. Note that the *state_handle* parameter is not used in this example, and the DRBG's entropy source for this example is used as the randomness source. The *prediction_resistance_request* parameter is not included as an input parameter of the **Reseed_function** for this example, since the entropy source will provide fresh entropy by definition.

The **Reseed_function** uses a **Get_entropy_input** call to obtain entropy bits from the entropy source. The **Get_entropy_input** call is fulfilled using the construction in Section 10.3.3.3. The **Get_entropy_input** call within the **Reseed_function** is the same as that used for instantiation (see Appendix A.1.1).

A.2 Example Using an Oversampling Construction

The NRBG Oversampling construction is specified in Section 9.4, and requires an entropy source and a DRBG mechanism (see the left half of Figure A-2). A separate instantiation of the same DRBG mechanism will be used for direct DRBG access (see the right half of Figure A-2); this instantiation is, in effect, a separate DRBG.

The CTR_DRBG specified in SP 800-90A will be used as the DRBG mechanism, with AES-256 used as the underlying block cipher for the DRBG. The DRBG mechanism will use the block-cipher derivation function in SP 800-90A. The entire NRBG is contained within a single cryptographic module.

As specified in Section 9.4, a DRBG used as part of the NRBG must be instantiated (and reseeded) at the highest security strength possible for the implemented DRBG mechanism. Since AES-256 will be used as the underlying block cipher, the highest security strength that can be supported by the DRBG mechanism is 256 bits. Therefore, the DRBG instantiation used in the NRBG construction will be instantiated and reseeded at a 256-bit security strength.

The DRBG instantiation used for direct DRBG access will be instantiated at a security strength of 256 bits (the same as the DRBG instantiation used as part of the NRBG) using the entropy source within the NRBG as the randomness source. Note that other examples could select a different security strength for this DRBG instantiation and a different randomness source.

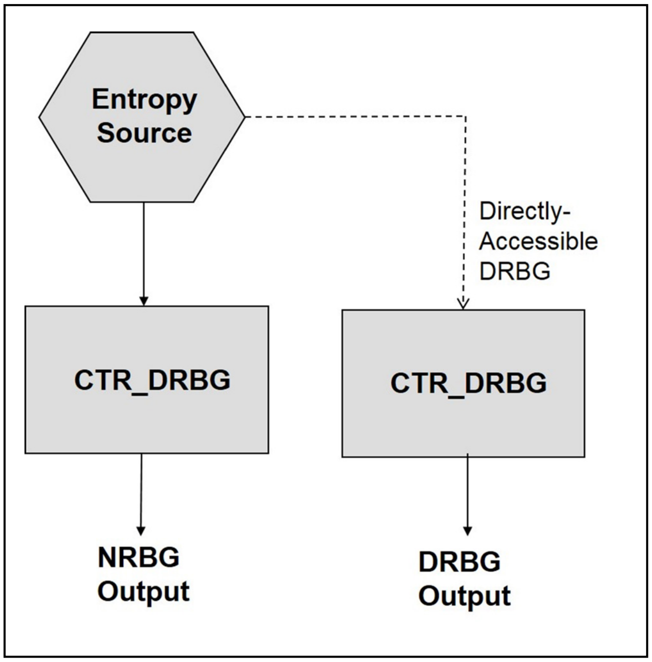

Figure A-2: NRBG Oversampling Construction Example

Calls are made to the NRBG using the NRBG calls specified in Section 7.3. Calls made to the directly accessible DRBG use the DRBG calls specified in Section 7.2.

The NRBG's DRBG supports prediction resistance by design (see Section 9.4). For this example, since a Live Entropy Source is always available, the directly accessed DRBG will also support prediction resistance.

As in the case of the XOR example in Appendix A.1, if the entropy source produces output at a slow rate, a consuming application might call the NRBG only when full entropy bits are required, obtaining all other output from the directly accessed DRBG.

This example provides the following capabilities:

- Full entropy output by the NRBG,
- Fallback to the security strength of the NRBG's DRBG (256 bits) if the entropy source has an undetected failure,
- Direct access to a DRBG for faster output,
- Both DRBGs instantiated at a security strength of 256 bits,
- Access to a Live Entropy Source to instantiate and reseed both DRBG instantiations, and
- Prediction resistance support for the directly accessed DRBG[5].

[5] Note that the prediction resistance provided by the NRBG's DRBG is not specifically listed, since it is

A.2.1 NRBG Instantiation

NRBG instantiation includes the instantiation of the DRBG in the NRBG construction (see Section 9.4.1). For this example, the DRBG mechanism will be instantiated twice: once for its use in the NRBG, and once for its use as a DRBG that is directly accessible using the DRBG calls in Section 7.2. If a success is not returned from either instantiation request, an invalid state handle (i.e., −1) will be returned. Note that the construction in Section 9.4.1 has been used as the basis for the following modified construction.

The **Modified_NRBG_Instantiate** construction is:

Modified_NRBG_Instantiate:

Input: bitstring *personalization_string*.

Output: integer *status*, integer *NRBG_state_handle, DRBG_state_handle*.

Process:

Comment: For step 1, *NRBG_state_handle* is the DRBG state handle when the DRBG mechanism is used as a component of the NRBG (i.e., the DRBG instantiation is not called directly by a consuming application using DRBG calls).

1. (*status*, *NRBG_state_handle*) = **Instantiate_function**(256, *prediction_resistance_flag* = TRUE, "NRBG" || *personalization_string*).

2. If (*status* ≠ SUCCESS), then return (*status*, −1, −1).

Comment: For step 3, *DRBG_state_handle* is the DRBG state handle when the DRBG instantiation is accessed using DRBG calls by a consuming application.

3. (*status*, *DRBG_state_handle*) = **Instantiate_function**(256, *prediction_resistance_flag* = TRUE, "DRBG" || *personalization_string*).

4. If (*status* ≠ SUCCESS), then return (*status*, −1, −1).

5. Return SUCCESS, *NRBG_state_handle, DRBG_state_handle*.

Note that the *requested_security_strength* parameter has been set to 256 bits for both DRBG instantiations, and a different string has been prepended to the personalization string to make them different for each instantiation (see steps 1 and 3). If there are no errors, and the entropy bits are available (as checked in steps 2 and 4), two different state handles are returned from the **Instantiate_function** calls. Also, since prediction resistance will be used during **NRBG_Generate** calls (see Section 9.4.1) and will be supported during direct accesses of the DRBG, the *prediction_resistance_flag* is set to TRUE during both **Instantiate_function** calls, rather than provided as input during the **Modified_NRBG_Instantiate** call. The **Instantiate_function** is specified in SP 800-90A.

included by design in bullet 1.

During the **Instantiate_function** calls, a **Get_entropy_input** call will be invoked to obtain entropy bits to instantiate the DRBG mechanism. The **Get_entropy_input** call is:

(*status*, *returned_bits*) **= Get_entropy_input** (256, 512),

which is fulfilled using the construction in Section 10.3.3.1. In this call, the *min_entropy* parameter is set to 256; the *max_length* parameter is set to an implementation-dependent value, say 512 for this example; and the *prediction_resistance_request* parameter is not used in this example, because the entropy source provides fresh entropy bits by design.

A.2.2 NRBG Generation

The NRBG can be called by a consuming application to generate output with full entropy. The construction in Section 9.4.2 is used:

NRBG_Generate:

Input: integer (*state_handle*, *n*), string *additional_input*.

Output: integer *status*, bitstring *returned_bits*.

Process:

1. *returned_bits* =*Null*.
2. *sum* = *0.*
3. While (*sum* < *n*)
   - 3.1 (*status*, *tmp*) = **Generate_function**(*NRBG_state_handle,* 128, 256, *prediction_resistance_request* = TRUE, *additional_input*).
   - 3.2 If (*status* ≠ SUCCESS), then return *status*, *Null*.
   - 3.3 *returned_bits* = *returned_bits* || *tmp.*
   - 3.4 *sum* = *sum* + 128.
4. Return SUCCESS and **leftmost**(*returned_bits*, *n*).

For this example, the NRBG's DRBG has been instantiated at 256 bits (see Appendix A.2.1); therefore, the security strength *s* = 256. Step 3.1 requests that the NRBG generate 128 bits (i.e., *s*/2 bits) at a security strength of 256 bits with prediction resistance; this will result in 128 bits of full-entropy output for each **Generate_function** call (see Sections 5.2 and 9.4.2). Note that the value of the state handle returned during the instantiation of the NRBG's DRBG instantiation is used in the **Generate_function** call, not the state handle that can be used by a consuming application to make calls directly to the DRBG.

During each execution of the **Generate_function** (i.e., for each 128-bit block of output produced by the **Generate_function**), the entropy source will be requested using the **Get_entropy_input** construction in Section 10.3.3.1.

A.2.3 Direct DRBG Generation

The DRBG instantiation used for direct access can be accessed by a consuming application using the **Generate_function** call specified in Section 7.2 as follows:

(*status*, *returned_bits*) = **Generate_function**(*DRBG_state_handle*, *n*, *security_strength*, *prediction_resistance_request*, *additional_input*).

Note that the *DRBG_state_handle* parameter is the value returned during instantiation for direct access of the DRBG mechanism by a consuming application (see Appendix A.2.1). A request for prediction resistance is optional, and the NRBG's entropy source is the randomness source for any prediction resistance request. The *security_strength* parameter must be less than or equal to 256 for this example.

When prediction resistance is requested in the **Generate_function** call, a single **Reseed_function** request will be made to the entropy source to produce a bitstring containing at least 256 bits of entropy (i.e., the security strength of the directly accessed DRBG), regardless of the number of bits (*n*) requested from the DRBG by the consuming application. This request is discussed in Appendix A.2.4.

A.2.4 Direct DRBG Reseeding

The DRBG instantiation that is directly accessible by a consuming application will be reseeded 1) if explicitly requested by the consuming application, 2) automatically whenever a generation with prediction resistance is requested during a direct access of the DRBG, or 3) automatically during a **Generate_function** call at the end of the DRBG's designed *reseed_interval* (see the **Generate_function** specification in SP 800-90A). The **Reseed_function** call is:

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

Note the specification of the *DRBG_state_handle*. The **Reseed_function** uses the **Get_entropy_input** call specified in Section 10.3.3.1.

The *prediction_resistance_request* parameter is omitted in the **Reseed_function** call for this example, since the randomness source is an entropy source.

A.3 Example Using a DRBG without a Randomness Source

A DRBG may have access to a randomness source only during instantiation (e.g., the DRBG will not have access to a Live Entropy Source or a source RBG during normal operation). For example, this will often be the case for smart card applications. In this case, the DRBG is seeded only once (i.e., reseeding is not possible).

For this example, the DRBG is distributed into two cryptographic modules, with a secure channel connecting them during the instantiation process; following DRBG instantiation, the secure channel is not available. The randomness source is an **approved** entropy source, no external conditioning function is used, and only a single DRBG instantiation will be used (see Figure A-3).

The DRBG will be instantiated at a *security_strength* of 256 bits, so a DRBG mechanism that is able to support this security strength must be used (e.g., HMAC_DRBG using SHA-256). A *personalization_string* will not be used. Since a randomness source is not available during normal operation, reseeding and prediction resistance cannot be provided.

This example provides the following capability:

- A DRBG instantiated at a security strength of 256 bits.

A.3.1 DRBG Instantiation

The DRBG is instantiated as specified in SP 800-90A using the following call:

status = **Instantiate_function** (256).

Note that since there will be only a single instantiation, a *state_handle* will not be returned for this example. In addition, a *prediction_resistance_flag* is not included, since a Live Entropy Source is not available after instantiation, so prediction resistance cannot be provided.

The **Instantiate_function**'s **Get_entropy_input** call is fulfilled using the construction in Section 10.3.3.1.

(*status*, *returned_bits*) = **Get_entropy_input**(256, 600),

This call sets the values of *min_entropy* to 256 bits, and *max_length* to 600 bits.

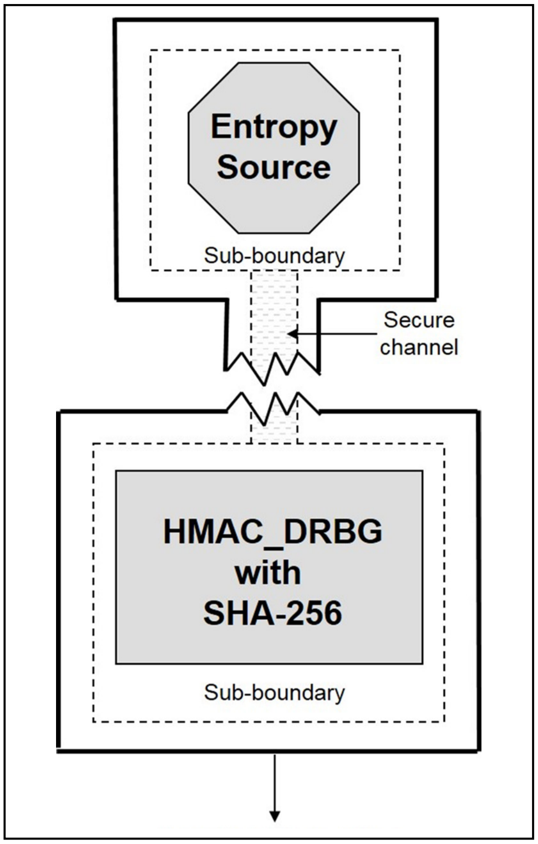

Figure A-3: DRBG Seeded Only Once

A secure channel is required to transport the entropy bits from the entropy source to the DRBG mechanism during instantiation. Thereafter, the entropy source and secure channel are no longer available (i.e., the connection between the entropy source and the DRBG mechanism is no longer available).

The *status* returned by the **Instantiate_function should** be checked; if a *status* of SUCCESS is not returned, then the DRBG has not been instantiated and cannot be used to generate (pseudo) random bits.

A.3.2 DRBG Generation

Pseudorandom bits are requested from the DRBG by a consuming application using the **Generate_function** call as specified in Section 7.2:

(*status*, *returned_bits*) = **Generate_function** (*requested_number_of_bits*, *requested_security_strength*, *additional_input*).

Since the instantiate call does not return a *state_handle* (see Appendix A.3.1), the *state_handle* parameter is not included in the generate request. The *requested_security_strength* may be any value that is less than or equal to 256 (the instantiated security strength). Since a Live Entropy Source will not be available, the *prediction_resistance_request* parameter is also omitted.

A.3.3 DRBG Reseeding

Since a randomness source is not available for reseeding, the DRBG must cease operation at the end of its designed *reseed_interval*. However, since the *reseed_interval* could be very long (up to 2^{48} requests, depending on the implementation), this may not be a problem for many applications.

A.4 Example Using a DRBG with a Live Entropy Source

A DRBG with a Live Entropy Source can provide prediction resistance on request. The entropy source could reside in the same device as the DRBG, or could reside outside the device, with a secure channel available to transfer the requested entropy bits to the DRBG mechanism (i.e., the DRBG is distributed).

For this example, assume that everything is the same as the example in Appendix A.3, except that a Live Entropy Source is available within the same cryptographic module as the DRBG mechanism. That is, the randomness source is an **approved** entropy source, no secure channel is required, and only a single DRBG instantiation will be used. The DRBG will be instantiated at a *security_strength* of 256 bits, so a DRBG mechanism that can support this security strength must be used (e.g., HMAC_DRBG using SHA-256). A *personalization_string* will not be used. Since a Live Entropy Source is available during normal operation, prediction resistance and reseeding are supported. Figure A-4 depicts this example.

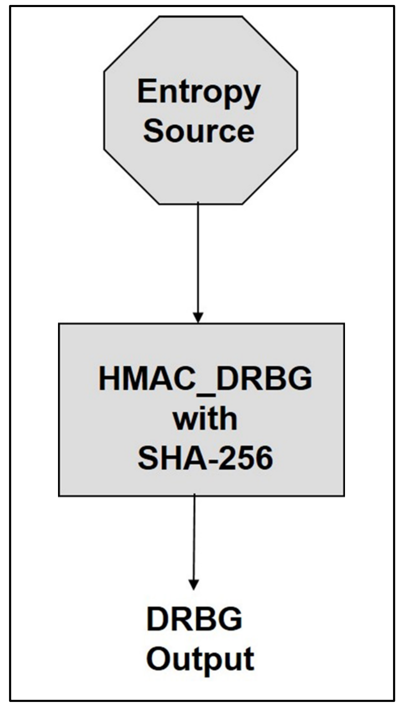

Figure A-4: DRBG with a Live Entropy Source

This example provides the following capabilities:

- Direct access to a DRBG,
- DRBG instantiated at a security strength of 256 bits,
- Access to a Live Entropy Source to provide prediction resistance and reseeding, and
- Full entropy output is possible.

A.4.1 DRBG Instantiation

The DRBG is instantiated as specified in SP 800-90A using the following call:

status = **Instantiate_function** (256, *prediction_resistance_flag*).

Note that since there will only be a single instantiation in this example, a *state_handle* will not be returned.

During the **Instantiate_function** call, a **Get_entropy_input** call using the construction in Section 10.3.3.1 will be invoked to obtain entropy bits to instantiate the DRBG mechanism. The **Get_entropy_input** call is:

(*status*, *returned_bits*) **= Get_entropy_input** (256, 512).

In the **Get_entropy_input** call, the *min_entropy* parameter is set to 256; the *max_length* parameter is set to an implementation-dependent value (i.e., 512 for this example).

The difference between the instantiation for this example, and the instantiation in Appendix A.3.1 is the inclusion of the *prediction_resistance_flag* in the **Instantiate_function** call. Note that a consuming application is not required to provide this parameter when calling the **Instantiate_function** unless prediction resistance is to be provided during normal operation when the DRBG is requested to generate bits (see Appendix A.4.2).

The consuming application **should** check the status returned by the **Instantiate_function**; if an indication of success is not returned, then the DRBG has not been instantiated and cannot be used to generate (pseudo) random bits.

A.4.2 DRBG Generation

Since a full-entropy capability is to be provided using an entropy source with no external conditioning function, the **General_DRBG_Generate** function discussed in Sections 7.2.2 and 10.4 will be used, i.e.,

(*status*, *returned_bits*) = **General_DRBG_Generate**(*requested_number_of_bits*, *security_strength*, *full_entropy_request*, *prediction_resistance_request*, *additional input*).

Since the instantiate call does not return a *state_handle* for this example (see Appendix A.4.1), the *state_handle* parameter is not included in the generate request. The *requested_security_strength* may be any value that is less than or equal to 256.

When full entropy or prediction resistance is requested, a **Get_entropy_input** call using the construction in Section 10.4 will be invoked to obtain entropy bits.

The consuming application **should** check the status returned by the **General_DRBG_Generate**_function; if an indication of success is not returned, then the requested bits have not been returned.

Note that the DRBG may need to be reseeded because of a prediction-resistance request or because of reaching the end of the DRBG's reseed interval, as discussed in Appendix A.4.3.

A.4.3 DRBG Reseeding

The DRBG will be reseeded 1) if explicitly requested by the consuming application, 2) automatically whenever generation with prediction resistance is requested, or 3) automatically during a **Generate_function** call at the end of the DRBG's designed reseed_interval (see the **Generate_function** specification in SP 800-90A). The **Reseed_function** call is:

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

The *state_handle* parameter has been omitted, since it is not required for this example. Note that the *prediction_resistance_request* parameter is omitted in the **Reseed_function** call, since fresh entropy bits are obtained from the entropy source anyway.

The **Get_entropy_input** call of the **Reseed_function** uses the construction in Section 10.3.3.1 to obtain entropy bits.

A.5 Example Using a Chain of DRBGs with a Live Entropy Source

Figure A-5 displays two chains of DRBGs, each with the same randomness source (i.e., both DRBG B and DRBG C have DRBG A as a randomness source). Each DRBG mechanism is contained within a different cryptographic module, and there is only one DRBG instantiation in each module. DRBG A has a Live Entropy Source as the randomness source that provides full-entropy output, but no external conditioning function. DRBG A is connected to DRBG B and DRBG C via secure channels. This configuration might be appropriate for a large organization that centralizes its initial DRBG of the chain (DRBG A, in this case) for use by other entities within the organization (e.g., each lower-level DRBG may be in a different employee's laptop).

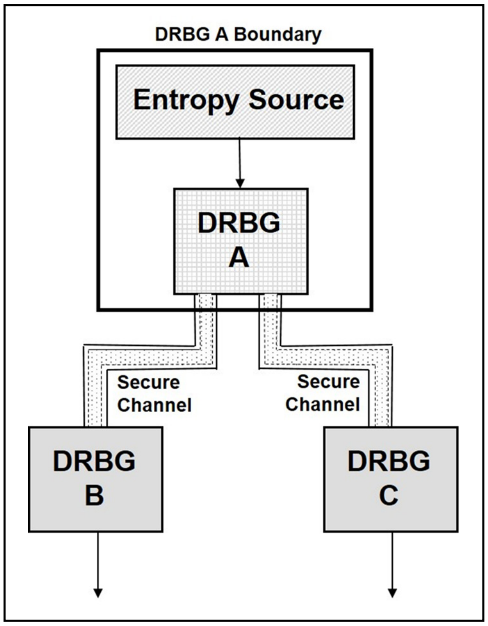

Figure A-5: Chain of DRBGs with a Live Entropy Source

The DRBGs may be implemented using the same or different DRBG mechanisms. This might be the case if the DRBGs are developed by different vendors. For simplicity in this example, the DRBG mechanisms are not shown.

For this example, DRBG A will be instantiated at a security strength of 128 bits and can provide prediction resistance when requested because a Live Entropy Source is always available. DRBG A will not be capable of handling a *personalization_string*.

DRBG B will be instantiated at a security strength of 128 bits, and DRBG C will be instantiated at a security strength of 256 bits; each will be capable of handling a *personization_string*. Each of the DRBG mechanisms (i.e., DRBGs A, B and C) allow a maximum of 512 bits to be input during a **Get_entropy_input** call (i.e., the *max_length* input parameter of the **Get_entropy_input** call must be less than or equal to 512).

This example provides the following capabilities:

- Direct access to each DRBG,
- DRBG A (the source DRBG) is instantiated at a security strength of 128 bits,
- DRBG B is instantiated at a security strength of 128 bits, while DRBG C is instantiated at a security strength of 256 bits,
- A Live Entropy Source is available to provide prediction resistance, and full-entropy output.

A.5.1 DRBG Instantiation

A.5.1.1 Instantiation of the Initial DRBG in the Chain (Source DRBG A)

For this example, DRBG A will be instantiated at a security strength of 128 bits using the following call (see SP 800-90A):

status = **Instantiate_function** (128, *prediction_resistance_flag* = TRUE).

Note that since there will only be a single instantiation, a *state_handle* will not be returned. The *prediction_resistance_flag* is set to TRUE to allow calls to DRBG A for prediction resistance (e.g., from DRBG B or C)). Also, note that there is no *personalization_string* parameter for this DRBG, as stated in Appendix A.5.

During the **Instantiate_function** call for this example, a **Get_entropy_input** call is fulfilled using the construction in Section 10.3.3.1 . The **Get_entropy_input** call is:

(*status*, *returned_bits*) **= Get_entropy_input** (128, 512).

In this call, the *min_entropy* parameter is set to 128, and the *prediction_resistance_request* parameter is omitted, since the entropy source is used directly.

The consuming application should check that the status returned by the **Instantiate_function**; if a *status* code of SUCCESS is not returned, then DRBG A has not been instantiated and cannot be used to generate random output (e.g., to service requests from DRBG B and DRBG C).

A.5.1.2 Instantiation of DRBG B

DRBG B is instantiated using the **Instantiate_function** call specified in SP 800-90A. The **Instantiate_function** call for requesting a security strength of 128 bits for DRBG B is:

status = **Instantiate_function** (128, *prediction_resistance_flag*, *personalization_string*).

Since only one DRBG instantiation is to be available in the device, the return of a *state_handle* is not required and has been omitted from the call.

During the instantiation of DRBG B, a request for output from DRBG A is made using a **Get_entropy_input** call in the **Instantiate_function**.

(*status*, *entropy_input*) = **Get_entropy_input(**128, 128, 512,
prediction_resistance_request =TRUE).

Since DRBG B is to be instantiated at the same security strength as DRBG A, the **Get_entropy_input** function can be implemented using either the construction in Section 10.1.1 or 10.1.2. In either case, the request for prediction resistance is optional, but for this example, prediction resistance is requested for instantiation.

Note that an implementation might combine the *min_entropy* and *min_length* parameters into a single parameter: the *security_strength*.

Upon receipt of this request from DRBG B, DRBG A generates output as discussed in Appendix A.5.2.1.

The consuming application **should** check the status returned by the **Instantiate_function**; if a *status* of SUCCESS is not returned, then DRBG B has not been instantiated and cannot generate (pseudo) random bits.

A.5.1.3 Instantiation of DRBG C

DRBG C is instantiated in the same manner as DRBG B, except that a security strength of 256 bits is required. The **Instantiate_function** call is:

status = **Instantiate_function** (256, *prediction_resistance_flag*, *personalization_string*).

Again, since only one DRBG instantiation is to be available in the device, the return of a *state_handle* is not required and has been omitted from the call.

The **Get_entropy_input** call in DRBG C's **Instantiate_function** in this case is:

(*status*, *entropy_input*) = **Get_entropy_input(**256, 256, 512, *prediction_resistance_request* = TRUE),

which requires the use of the **Get_entropy_input** construction in Section 10.1.2, since DRBG C is instantiating at a higher security strength than that of DRBG A.

DRBG A's handling of the received request is discussed in Appendix A.5.2.1.

The consuming application **should** check that the status returned by the **Instantiate_function**; if a *status* of SUCCESS is not returned, then DRBG C has not been instantiated and cannot be used to generate (pseudo) random bits.

A.5.2 DRBG Generation

A.5.2.1 Generate Requests to DRBG A from a Subsequent DRBG in a Chain

Generate requests to DRBG A are made by the subsequent DRBGs in the chain (i.e., DRBGs B and C) during instantiation or reseeding using the **Get_entropy_input** construction used in Appendix A.5.1.2 and A.5.1.3. A generate request is sent to DRBG A in the form of a **Generate_function** call, which will indicate the security strength to be used, the minimum and maximum length of the bitstring to be returned, and possibly a request for prediction resistance. As specified in SP 800-90A, when prediction resistance is requested, DRBG A reseeds itself by requesting a bitstring from its entropy source containing128 bits of entropy.

Generate requests may also be made directly to DRBG A by a consuming application (see Appendix A.5.2.2).

The reseeding of DRBG A is discussed in Appendix A.5.3.1.

A.5.2.2 Generate Requests to a DRBG by a Consuming Application

Generate requests could be made directly to any of the DRBGs in the chain from a consuming application, including requests to DRBG A. Since any of the DRBGs can be requested to provide full-entropy output, the **General_DRBG_Generate** function discussed in Sections 7.2.2 and 10.4 will be used, i.e.,

(*status*, *returned_bits*) = **General_DRBG_Generate**(*requested_number_of_bits*, *security_strength*, *full_entropy_request*, *prediction_resistance_request*, *additional input*).

Note that even though DRBG A's entropy source provides full-entropy output, DRBG A is designed to do so only when using the appropriate construction.

Since the instantiate call does not return a *state_handle* for this example (see Appendix A.5.1), the *state_handle* parameter is not included in the generate request. The *requested_security_strength* may be any value that is less than or equal to 256.

When full entropy or prediction resistance are requested, a **Get_entropy_input** call using the construction in Section 10.4 will be invoked by DRBG B and DRBG C to obtain entropy bits. DRBG A will use the **Get_entropy_input** construction in Section 10.3.3.3, which will provide full-entropy output.

The consuming application **should** check the status returned by the **General_DRBG_Generate**_function; if an indication of success is not returned, then the requested bits have not been returned.

Note that the DRBG may need to be reseeded because of a prediction-resistance request or because of reaching the end of the DRBG's reseed interval, as discussed in Appendix A.5.3.

A.5.3 DRBG Reseeding

A.5.3.1 Reseeding of DRBG A (the Initial DRBG of the Chain)

DRBG A can be reseeded using its **Reseed_function** to obtain entropy bits from its Live Entropy Source. The reseed of DRBG A is initiated because of a request for bits with prediction resistance from DRBG B or DRBG C, a reseed request to DRBG A directly from a consuming application, or reaching the end of the DRBG's reseed interval during a **Generate_function** call from a consuming application or a subsequent DRBG of a chain). The **Reseed_function** call for this example is:

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

The *state_handle* parameter has been omitted since it is not required for this example.

The **Reseed_function** in DRBG A makes a **Get_entropy_input** call to obtain the entropy input for reseeding from DRBG A's Live Entropy Source. The **Get_entropy _input** call is specified in Section 10.3.3.1, althought the construction in Section 10.3.3.2 or Section 10.3.3.3 could also be used.

When reseeding at the request of from a consuming application, the consuming application **should** check the status returned by the **Reseed_function**; if a *status* of SUCCESS is not returned, then the DRBG has not been reseeded.

A.5.3.2 Reseeding of a Subsequent DRBG in a Chain

DRBGs B and C are reseeded by requesting output from DRBG A. The reseed process is initiated because of a reseed request to the DRBG from a consuming application, a request from the consuming application for prediction resistance during a **Generate_ request**, or reaching the end of the DRBG's reseed interval during a **Generate_function** call from a consuming application).

The **Reseed_function** call for this example is:

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

The *state_handle* parameter has been omitted since it is not required for this example.

The **Reseed_function** makes a **Get_entropy_input** call to DRBG A to obtain the entropy input for reseeding. The **Get_entropy_input** function uses the same construction used for instantiation (see Appendix A.5.1.2 for DRBG B, and Appendix A.5.1.3 for DRBG C).

If the **Reseed_function** is called by the consuming application, the call has the same form as above. However, the presence of a *prediction_resistance_request* parameter in the subsequent **Get_entropy_input** call depends on its presence in the **Reseed_function** call from the consuming application. The consuming application **should** check that the status returned by the **Reseed_function**; if a *status* of SUCCESS is not returned, then the DRBG has not been reseeded.

If the call is initiated from within DRBG B, a request for prediction resistance is optional, since DRBG A's security strength is the same as that of DRBG B. However, if the call is initiated from within DRBG C, a prediction-resistance request is required, since DRBG A's security strength is less than that of DRBG C; this is handled in the **Get_entropy_input** routine used by DRBG C (i.e., the routine specified in Section 10.1.2).

Appendix B: References

[FIPS 140] Federal Information Processing Standard (FIPS) 140-2, Security Requirements for Cryptographic Modules, May 2001.

[FIPS 180] Federal Information Processing Standard (FIPS) 180-4, Secure Hash Standard, March 2012.

[FIPS 197] Advanced Encryption Standard (AES), November 2001, available at http://csrc.nist.gov/publications/PubsFIPS.html.

[FIPS 198] Federal Information Procssing Standard (FIPS) 198-1, The Keyed-Hash Message Authentication Code (HMAC), July 2008.

[FIPS 202] Federal Information Processing Standard (FIPS) 202, DRAFT SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, August 2015.

[SP 800-38B] NIST Special Publication (SP) 800-38B, Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication, May 2005.

[SP 800-57] NIST Special Publication (SP) 800-57, Part 1: Recommendation for Key Management: Part 1: General (Revision 3), January 2016.

[SP 800-67] NIST Special Publication (SP) 800-67 Rev. 1, Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher, January 2012.

[SP 800-90A] NIST Special Publication 800-90A, Recommendation for Random Number Generation Using Deterministic Random Bit Generators, June 2015.

[SP 800-90B] NIST Special Publication 800-90B, (Draft) Recommendation for the Entropy Sources Used for Random Bit Generation, January 2016.

[SP 800-107] NIST Special Publication 800-107, Recommendation for Applications Using Approved Hash Algorithms, August 2012.

[ANS X9.82-4] Random Number Generation - Part 4: Random Bit Generation Constructions, April 2011.

[ILL89] R. Impagliazzo, L. A. Levin, and M. Luby. *Pseudo-random generation from one-way functions*. In Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC '89), pages 12-24. ACM Press, 1989.