NIST Special Publication
NIST SP 800-90C 3pd

Recommendation for Random Bit Generator (RBG) Constructions

Third Public Draft (3pd)

Elaine Barker
John Kelsey
Kerry McKay
Allen Roginsky
Meltem Sönmez Turan

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-90C.3pd

NATIONAL INSTITUTE OF
STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

NIST Special Publication
NIST SP 800-90C 3pd

Recommendation for Random Bit Generator (RBG) Constructions

Third Public Draft (3pd)

Elaine Barker
John Kelsey
Kerry McKay
Allen Roginsky
Meltem Sönmez Turan
Computer Security Division
Information Technology Laboratory

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-90C.3pd

September 2022

Authority

NIST Technical Series Policies

Publication History

Approved by the NIST Editorial Review Board on YYYY-MM-DD [will be added in final published version]

How to Cite this NIST Technical Series Publication:

Barker EB, Kelsey JM, McKay KA, Roginsky AL, Sönmez Turan M (2022) Recommendation for Random Bit Generator (RBG) Constructions. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-90C 3pd. https://doi.org/10.6028/NIST.SP.800-90c.3pd

NIST Author ORCID iDs [will be added in final published version]

Author 1: 0000-0000-0000-0000
Author 2: 0000-0000-0000-0000
Author 3: 0000-0000-0000-0000
Author 4: 0000-0000-0000-0000

Public Comment Period
September 7, 2022 – December 7, 2022

Submit Comments
rbg_comments@nist.gov

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

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

Reports on Computer Systems Technology

Abstract

The NIST Special Publication (SP) 800-90 series of documents supports the generation of high-quality random bits for cryptographic and non-cryptographic use. SP 800-90A specifies several deterministic random bit generator (DRBG) mechanisms based on cryptographic algorithms. SP 800-90B provides guidance for the development and validation of entropy sources. This document (SP 800-90C) specifies constructions for the implementation of random bit generators (RBGs) that include DRBG mechanisms as specified in SP 800-90A and that use entropy sources as specified in SP 800-90B. Constructions for three classes of RBGs (namely, RBG1, RBG2, and RBG3) are specified in this document.

Keywords

deterministic random bit generator (DRBG); entropy; entropy source; random bit generator (RBG); randomness source; RBG1 construction; RBG2 construction; RBG3 construction; subordinate DRBG (sub-DRBG).

Note to Reviewers

1. This draft of SP800-90C describes three RBG constructions. Note that in this draft, a non-deterministic random bit generator (NRBG) is presented as an RBG3 construction.

   Question: *In a future revision of SP 800-90C, should other constructions be included?*

   This version of SP 800-90C does not address the use of an RBG software implementation in which a) a cryptographic library or an application is loaded into a system and b) the software accesses entropy sources or RBGs already associated with the system for its required randomness. NIST intends to address this situation in the near future.

2. The RBG constructions provided in this draft use NIST-**approved** cryptographic primitives (such as block ciphers and hash functions) as underlying components. Note that non-vetted conditioning components may be used within SP 800-90B entropy sources.

   Although NIST still allows three-key TDEA as a block-cipher algorithm, Section 4 of [SP800-131A] indicates that its use is deprecated through 2023 and will be disallowed thereafter for applying cryptographic protection. This document (i.e., SP 800-90C) **does not approve** the use of three-key TDEA in an RBG.

   Although SHA-1 is still approved by NIST, NIST is planning to remove SHA-1 from a future revision of FIPS 180-4, so the SP 800-90 series will not be including the use of SHA-1.

   The use of the SHA-3 hash functions are **approved** in SP 800-90C for Hash_DRBG and HMAC_DRBG but are not currently included in [SP800-90A]. SP 800-90A will be revised to exclude the use of TDEA and SHA-1 and include the use of the SHA-3 family of hash functions.

3. Since the projected date for requiring a minimum security strength of 128 bits for U.S. Government applications is 2030 (see [SP800-57Part1]), RBGs are only specified to provide 128, 192, and 256 bits of security strength (i.e., the 112-bit security strength has been removed). Note that a consuming application may still request a lower security strength, but the RBG output will be generated at the instantiated security strength.

4. Guidance is provided for accessing entropy sources and for obtaining full-entropy bits using the output of an entropy source that does not inherently provide full-entropy output (see Section 3.3).

5. SP 800-90A requires that when instantiating a CTR_DRBG without a derivation function, the randomness source needs to provide full-entropy bits (see SP 800-90A). However, this draft (SP 800-90C) relaxes this requirement in the case of an RBG1 construction, as specified in Section 4. In this case, the external randomness source may be another RBG construction. An addendum to SP 800-90A has been prepared as a temporary specification in SP 800-90C, but SP 800-90A will be revised in the future to accommodate this change.

6. The DRBG used in RBG3 constructions supports a security strength of 256 bits. The RBG1 and RBG2 constructions may support any valid security strength (i.e., 128, 192 or 256 bits).

7. SP 800-90A currently allows the acquisition of a nonce (when required) for DRBG instantiation from any randomness source. However, SP 800-90C does not include an explicit requirement for the generation of a nonce when instantiating a DRBG. Instead, additional bits

beyond those needed for the security strength are acquired from the randomness source. SP 800-90A will be revised to agree with this change.

8. SP 800-90C allows the use of both physical and non-physical entropy sources. See the definitions of physical and non-physical entropy sources in Appendix E. Also, multiple validated entropy sources may be used to provide entropy, and two methods are provided in Section 2.3 for counting the entropy provided in a bitstring.

9. The CMVP is considering providing information on an entropy source validation certificate that indicates whether an entropy source is physical or non-physical.

10. The CMVP is developing a program to validate entropy sources against SP 800-90B with the intent of allowing the re-use of those entropy sources in different RBG implementations.

    Question: *Are there any issues that still need to be addressed in SP 800-90C to allow the re-use of validated entropy sources in different RBG implementations? Note that in many cases, specific issues need to be addressed in the FIPS 140 implementation guide rather than in this document.*

Call for Patent Claims

This public review includes a call for information on essential patent claims (claims whose use would be required for compliance with the guidance or requirements in this Information Technology Laboratory (ITL) draft publication). Such guidance and/or requirements may be directly stated in this ITL Publication or by reference to another publication. This call also includes disclosure, where known, of the existence of pending U.S. or foreign patent applications relating to this ITL draft publication and of any relevant unexpired U.S. or foreign patents.
ITL may require from the patent holder, or a party authorized to make assurances on its behalf, in written or electronic form, either:

a) assurance in the form of a general disclaimer to the effect that such party does not hold and does not currently intend holding any essential patent claim(s); or

b) assurance that a license to such essential patent claim(s) will be made available to applicants desiring to utilize the license for the purpose of complying with the guidance or requirements in this ITL draft publication either:

    i. under reasonable terms and conditions that are demonstrably free of any unfair discrimination; or

    ii. without compensation and under reasonable terms and conditions that are demonstrably free of any unfair discrimination.

Such assurance shall indicate that the patent holder (or third party authorized to make assurances on its behalf) will include in any documents transferring ownership of patents subject to the assurance, provisions sufficient to ensure that the commitments in the assurance are binding on the transferee, and that the transferee will similarly include appropriate provisions in the event of future transfers with the goal of binding each successor-in-interest.

The assurance shall also indicate that it is intended to be binding on successors-in-interest regardless of whether such provisions are included in the relevant transfer documents.

Such statements should be addressed to: rbg_comments@nist.gov

Table of Contents

List of Tables

List of Figures

Acknowledgments

The National Institute of Standards and Technology (NIST) gratefully acknowledges and appreciates contributions from Chis Celi (NIST); Darryl Buller, Aaron Kaufer, and Mike Boyle (National Security Agency); Werner Schindler, Matthias Peter, Johannes Mittman (Bundesamt für Sicherheit in der Informationstechnik); and the members of the Cryptographic Module User Forum (CMUF) for assistance in the development of this Recommendation. NIST also thanks the many contributions by the public and private sectors.

1. Introduction and Purpose

Cryptography and security applications make extensive use of random bits. However, the generation of random bits is challenging in many practical applications of cryptography.

The National Institute of Standards and Technology (NIST) developed the Special Publication (SP) 800-90 series to support the generation of high-quality random bits for both cryptographic and non-cryptographic purposes. The SP 800-90 series consists of three parts:

- SP 800-90A, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, specifies several **approved** deterministic random bit generator (DRBG) mechanisms based on **approved** cryptographic algorithms that – once provided with seed material that contains sufficient entropy – can be used to generate random bits suitable for cryptographic applications.
- SP 800-90B, *Recommendation for the Entropy Sources Used for Random Bit Generation*, provides guidance for the development and validation of entropy sources − mechanisms that generate entropy from physical or non-physical noise sources and that can be used to generate the input for the seed material needed by a DRBG or for input to an RBG.
- SP 800-90C, *Recommendation for Random Bit Generator (RBG) Constructions,* specifies constructions for random bit generators (RBGs) using entropy sources that comply with SP 800-90B and DRBGs that comply with SP 800-90A. Three classes of RBGs are specified in this document (see Sections 5, 6, and 7). SP 800-90C also provides high-level guidance for testing RBGs for conformance to this Recommendation.

The RBG constructions defined in this Recommendation consist of two main components: the *entropy sources* that generate true random variables (variables that may be biased, i.e., each possible outcome does not need to have the same chance of occurring) and the DRBGs that ensure that the outputs of the RBG are indistinguishable from the ideal distribution to a computationally bounded adversary.

Throughout this document, the phrase "this Recommendation" refers to the aggregate of SP 800-90A, SP 800-90B, and SP 800-90C, while the phrase "this document" refers only to SP 800-90C.

SP 800-90C has been developed in coordination with NIST's Cryptographic Algorithm Validation Program (CAVP) and Cryptographic Module Validation Program (CMVP). The document uses "**shall**" and "**must**" to indicate requirements and uses "**should**" to indicate an important recommendation. The term "**shall**" is used when a requirement is testable by a testing lab during implementation validation using operational tests or a code review. The term "**must**" is used for requirements that may not be testable by the CAVP or CMVP. An example of such a requirement is one that demands certain actions and/or considerations from a system administrator. Meeting these requirements can be verified by a CMVP review of the cryptographic module's documentation. If the requirement is determined to be testable at a later time (e.g., after SP 800-90C is published and before it is revised), the CMVP will so indicate in the Implementation Guidance for FIPS 140, *Security Requirements for Cryptographic Modules*.

1.1. Audience

The intended audience for this Recommendation includes 1) developers who want to design and implement RBGs that can be validated by NIST's CMVP and CAVP, 2) testing labs that are accredited to perform the validation tests and the evaluation of the RBG constructions, and 3) users who install RBGs in systems.

1.2. Document Organization

This document is organized as follows:

- Section 2 provides background and preliminary information for understanding the remainder of the document.
- Section 3 provides guidance on accessing and handling entropy sources, including the external conditioning of entropy-source output.
- Sections 4, 5, and 6 specify the RBG constructions.
- Section 7 discusses health and implementation-validation testing.
- References contains a list of papers and publications cited in this document.

The following informational appendices are also provided:

- Appendix A provides discussions on entropy versus security strength.
- Appendix B provides examples of each RBG construction.
- Appendix C is an addendum to SP 800-90A that includes two additional derivation functions that may be used with the CTR_DRBG. These functions will be moved into SP 800-90A as part of the next revision of that document.
- Appendix D provides a list of abbreviations, symbols, functions, and notations used in this document.
- Appendix E provides a glossary with definitions for terms used in this document.

2. General Information

2.1. RBG Security

Ideal randomness sources generate identically distributed and independent uniform random bits that provide full-entropy outputs (i.e., one bit of entropy per output bit). Real-world RBGs are designed with a security goal of *indistinguishability* from the output of an ideal randomness source. That is, given some limits on an adversary's data and computing power, it is expected that there is no adversary that can reliably distinguish between RBG outputs and outputs from an ideal randomness source.

Consider an adversary that can perform 2^w computations (typically, these are guesses of the RBG's internal state) and is given an output sequence from either an RBG with a security strength of s bits (where $s \geq w$) or an ideal randomness source. It is expected that an adversary has no better probability of determining which source was used for its random bits than

$$1/2 + 2^{w-s-1} + \varepsilon,$$

where ε is negligible. In this Recommendation, the size of the output is limited to 2^{64} output bits and $\varepsilon \leq 2^{-32}$.

An RBG that has been designed to support a security strength of s bits is suitable for any application with a targeted security strength that does not exceed s. An RBG that is compliant with this Recommendation can support requests for output with a security strength of 128, 192, or 256 bits, except for an RBG3 construction (as described in Section 6), which can provide full-entropy output.

A bitstring with full entropy has an amount of entropy equal to its length. Full-entropy bitstrings are important for cryptographic applications, as these bitstrings have ideal randomness properties and may be used for any cryptographic purpose. They may be truncated to any length such that the amount of entropy in the truncated bitstring is equal to its length. However, due to the difficulty of generating and testing full-entropy bitstrings, this Recommendation assumes that a bitstring has full entropy if the amount of entropy per bit is at least $1 - \varepsilon$, where ε is at most 2^{-32}. NISTIR 8427[1] provides a justification for the selection of ε.

2.2. RBG Constructions

A *construction* is a method of designing an RBG or some component of an RBG to accomplish a specific goal. Three classes of RBG constructions are defined in this document: RBG1, RBG2, and RBG3 (see Table 1). Each RBG includes a DRBG from [SP800-90A] and is based on the use of a randomness source that is validated for compliance with [SP800-90B] or SP 800-90C. Once instantiated, a DRBG can generate output at a security strength that does not exceed the DRBG's instantiated security strength.

[1] See NISTIR 8427, Discussion on the Full Entropy Assumption of SP 800-90 series.

Table 1. RBG Capabilities

Construction	Internal Entropy Source	Prediction Resistance	Full Entropy	Type of randomness source
RBG1	No	No	No	Physical
RBG2	Yes	Yes[a]	No	Physical or Non-physical
RBG3	Yes	Yes[a]	Yes	Physical

[a] If sufficient entropy is available or can be obtained when reseeding the RBG's DRBG.

1. An RBG1 construction (see Section 4) does not have access to a randomness source after instantiation. It is instantiated once in its lifetime over a secure channel from an external RBG with appropriate security properties. An RBG1 construction does not support reseeding and cannot provide *prediction resistance* as described in Section 2.4.2 and [SP800-90A]. The construction can be used to initialize subordinate DRBGs.

2. An RBG2 construction (see Section 5) includes one or more entropy sources that are used to instantiate and reseed the DRBG within the construction. This construction can provide prediction resistance (see Section 2.4.2 and [SP800-90A]) when sufficient entropy is available or can be obtained from the RBG's entropy source(s) at the time that prediction resistance is requested. The construction has two variants that depend on the type of entropy source(s) employed (i.e., physical and non-physical).

3. An RBG3 construction is designed to provide output with a security strength equal to the requested length of its output by producing outputs that have full entropy (i.e., an RBG designed as an RBG3 construction can, in effect, support all security strengths) (see Section 2.1). This construction provides prediction resistance and has two types, namely RBG3(XOR) and RBG3(RS).

   a. An RBG3(XOR) construction (see Section 6.2) combines the output of one or more validated entropy sources with the output of an instantiated, **approved** DRBG using an exclusive-or (XOR) operation.

   b. An RBG3(RS) construction (see Section 6.3) uses one or more validated entropy sources to provide randomness input for the DRBG by continuously reseeding.

This document also provides constructions for 1) subordinate DRBGs (sub-DRBGs) that are instantiated and possibly reseeded by an RBG1 construction (see Section 4.3) and 2) acquiring entropy from an entropy source and conditioning the output to provide a bitstring with full entropy (see Section 3.3). SP 800 90A provides constructions for instantiating and reseeding DRBGs and requesting the generation of pseudorandom bitstrings.

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

2.3. Sources of Randomness for an RBG

The RBG constructions specified in this document are based on the use of validated entropy sources. Some RBG constructions (e.g., the RBG3 construction) access these entropy sources directly to obtain entropy. Other constructions (e.g., the RBG1 construction) fulfill their entropy requirements by accessing another RBG as a randomness source. In this case, the source RBG may include its own entropy source.

SP 800 90B provides guidance for the development and validation of entropy sources – mechanisms that provide entropy for an RBG. Validated entropy sources (i.e., entropy sources that have been successfully validated by the CMVP as complying with SP 800-90B) provide fixed-length outputs and have been validated as reliably providing a specified minimum amount of entropy for each output (e.g., each eight-bit output has been validated as providing at least five bits of entropy).[2]

An entropy source is a *physical entropy source* if the primary noise source of the entropy source is physical – that is, it uses dedicated hardware to provide entropy (e.g., from ring oscillators, thermal noise, shot noise, jitter, or metastability). Similarly, a validated entropy source is a *non-physical entropy source* if the primary noise source of the entropy source is non-physical – that is, entropy is provided by system data (e.g., the entropy present in the RAM data or system time). The entropy-source type is certified during SP 800-90B validation.

One or more validated entropy sources are used to provide entropy for instantiating and reseeding the DRBGs in RBG2 or RBG3 constructions or used by an RBG3 construction to generate output upon request by a consuming application.

An implementation could be designed to use a combination of physical and non-physical entropy sources. When requests are made to the sources, bitstring outputs are concatenated until the amount of entropy in the concatenated bitstring meets or exceeds the request. Two methods are provided for counting the entropy provided in the concatenated bitstring.

Method 1: The RBG implementation includes one or more physical entropy sources, and one or more non-physical entropy sources may also be included in the implementation. However, only the entropy in a bitstring that is provided from physical entropy sources is counted toward fulfilling the amount of entropy requested in an entropy request. Any entropy in a bitstring that is provided by a non-physical entropy source is not counted, even if bitstrings produced by the non-physical entropy source are included in the concatenated bitstring that is used by the RBG.

Method 2: The RBG implementation includes one or more non-physical entropy sources, and one or more physical entropy sources may also be included in the implementation. The entropy from both non-physical entropy sources and (if present) physical entropy sources is counted when fulfilling an entropy request.

Example: Let pes_i be the ith output of a physical entropy source, and $npes_i$ be the jh output of a non-physical entropy source. If an implementation consists of one physical and one non-physical entropy source, and a request has been made for 128 bits of entropy, the concatenated bitstring might be something like:

$$pes_1 \,||\, pes_2 \,||\, npes_1 \,||\, pes_3 \,||\, \ldots \,||\, npes_m \,||pes_n,$$

which is the concatenated output of the physical and non-physical entropy sources.

[2] Note that this document also discusses the use of non-validated entropy sources. When discussing such entropy sources, "non-validated" will always precede "entropy sources." The use of the term "validated entropy source" may be shortened to just "entropy source" to avoid repetition.

According to Method 1, only the entropy in pes_1, pes_2, ..., pes_n would be counted toward fulfilling the 128-bit request. Any entropy in $npes_1$, ... $npes_m$ is not counted.

According to Method 2, all of the entropy in pes_1, pes_2, ... pes_n and in $npes_1$, $npes_2$, ..., $npes_m$ is counted. Since the entropy from both non-physical and physical entropy sources is counted in Method 2, the concatenated output string is expected to be shorter compared to that credited using Method 1.

When multiple entropy sources are used, there is no requirement on the order in which the entropy sources are accessed or the number of times that each entropy source is accessed to fulfill an entropy request (e.g., if two physical entropy sources are used, it is possible that a request would be fulfilled by only one of the entropy sources because entropy is not available at the time of the request from the other entropy source). However, the Method 1 or Method 2 criteria for counting entropy still applies.

This Recommendation assumes that the entropy produced by a validated physical entropy source is generally more reliable than the entropy produced by a validated non-physical entropy source since non-physical entropy sources are typically influenced by human actions or network events, the unpredictability of which is difficult to accurately quantify. Therefore, Method 1 is considered to provide more assurance that the concatenated bitstring actually contains at least the requested amount of entropy (128 bits for the example). Note that RBG2(P) and RBG3 constructions only count the entropy using Method 1 (see Sections 5 and 6).

2.4. DRBGs

Approved DRBG designs are specified in [SP800-90A]. A DRBG includes instantiate, generate, and health-testing functions and may include reseed and uninstantiate functions. The instantiation of a DRBG involves acquiring sufficient randomness to initialize the DRBG to support a targeted security strength and establish the internal state, which includes the secret information for operating the DRBG. The generate function produces output upon request and updates the internal state. Health testing is used to determine that the DRBG continues to operate correctly. Reseeding introduces fresh entropy into the DRBG's internal state and is used to recover from a potential (or actual) compromise (see Section 2.4.2 for additional discussion). An uninstantiate function is used to terminate a DRBG instantiation and destroy the information in its internal state.

2.4.1. DRBG Instantiations

A DRBG implementation consists of software code, hardware, or both hardware and software that is used to implement a DRBG design. The same implementation can be used to create multiple "copies" of the same DRBG (e.g., for different purposes) without replicating the software code or hardware. Each "copy" is a separate instantiation of the DRBG with its own internal state that is accessed via a state handle that is unique to that instantiation (see Figure 1). Each instantiation may be considered a different DRBG, even though it uses the same software code or hardware.

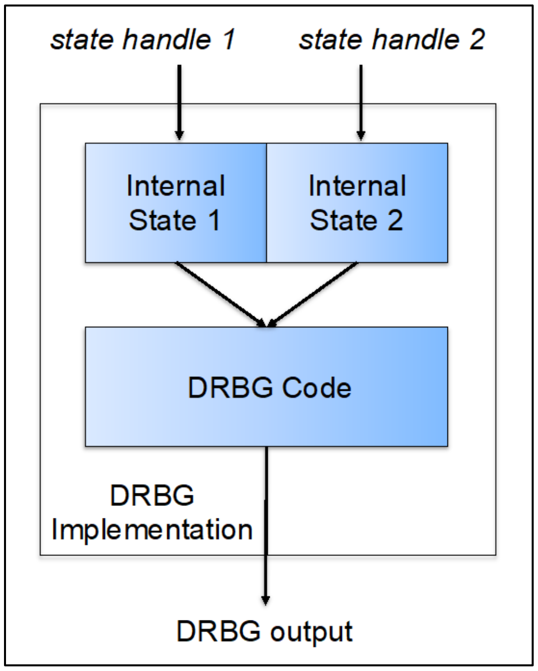

Fig. 1. DRBG Instantiations

Each DRBG instantiation is initialized with input from some randomness source that establishes the security strengths that can be supported by the DRBG. During this process, an optional but recommended personalization string may also be used to differentiate between instantiations in addition to the output of the randomness source. The personalization string could, for example, include information particular to the instantiation or contain entropy collected during system activity (e.g., from a non-validated entropy source). An implementation **should** allow the use of a personalization string. More information on personalization strings is provided in [SP800-90A].

A DRBG may be implemented to accept further input during operation from the randomness source (e.g., to reseed the DRBG) and/or additional input from inside or outside of the cryptographic module that contains the DRBG. This additional input could, for example, include information particular to a request for generation or reseeding or could contain entropy collected during system activity (e.g., from a validated or non-validated entropy source).[3]

2.4.2. DRBG Reseeding, Prediction Resistance, and Recovery from Compromise

Under some circumstances, the internal state of an RBG (containing the RBG's secret information) could be leaked to an adversary. This would typically happen as the result of a side-channel attack or tampering with a hardware device, and it may not be detectable by the RBG or any consuming application.

All DRBGs in [SP800-90A] are designed with *backtracking resistance* − that is, learning the DRBG's current internal state does not provide knowledge of previous outputs. Since all RBGs in SP 800-90C are based on the use of SP 800-90A DRBGs, they also inherit this property. However,

[3] Entropy provided in additional input does not affect the instantiated security strength of the DRBG instantiation. However, it is good practice to include any additional entropy when available to provide more security.

once the secret information within the DRBG's internal state is compromised, all future DRBG outputs are known to the adversary unless the DRBG is reseeded – a process that returns the DRBG to a non-compromised state.

A DRBG is reseeded when at least *s* bits of fresh entropy are used to update the internal state (where *s* is the security strength of the DRBG) so that the updated internal state is unknown and extremely unlikely to be correctly guessed. A DRBG that has been reseeded has *prediction resistance* against an adversary who knows its previous internal state. Reseeding may be performed upon request from a consuming application (either an explicit request for reseeding or a request for the generation of bits with prediction resistance); on a fixed schedule based on time, number of outputs, or events; or as sufficient entropy becomes available.

Although reseeding provides fresh entropy bits that are incorporated into an already instantiated DRBG at a security strength of *s* bits, this Recommendation does not consider the reseed process as increasing the DRBG's security strength. For example, a reseed of a DRBG that has been instantiated to support a security strength of 128 bits does not increase the DRBG's security strength to 256 bits when reseeding with 128 bits of fresh entropy.

An RBG1 construction has no access to a randomness source after instantiation and so cannot be reseeded or recover from a compromise (see Section 4). Thus, it can never provide prediction resistance.

An RBG2 construction contains an entropy source that is used to reseed the DRBG within the construction (see Section 5) and recover from a possible compromise of the RBG's internal state. Prediction resistance may be requested by a consuming application during a request for the generation of (pseudo) random bits. If sufficient entropy can be obtained from the entropy source(s) at that time, the DRBG is reseeded before the requested bits are generated. If sufficient entropy is not available, an error indication is returned, and no bits are generated for output. Therefore, it is recommended that prediction resistance not be claimed for an RBG implementation unless sufficient entropy is reliably available upon request.

An RBG3 construction is provided with fresh entropy for every RBG output (see Section 6). As a result, every output from an RBG3 construction has prediction resistance.

For a more complete discussion of backtracking and prediction resistance, see [SP800-90A].

2.5. RBG Security Boundaries

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

The primary components of an RBG are a randomness source (i.e., an entropy source or an RBG construction), a DRBG, and health tests for the RBG. RBG input (e.g., entropy bits and a personalization string) **shall** enter an RBG only as specified in the functions described in Section 2.8. The security boundary of a DRBG is discussed in [SP800-90A]. The security boundary for an entropy source is discussed in [SP800-90B]. Both the entropy source and the DRBG contain their own health tests within their respective security boundaries.

Figure 2 shows an RBG implemented within a [FIPS 140]-validated cryptographic module. The RBG security boundary **shall** either be the same as the cryptographic module boundary or be completely contained within that boundary. The data input may be a personalization string or additional input (see Section 2.4.1). The data output is status information and possibly random bits or a state handle. Within the RBG security boundary of the figure are an entropy source and a DRBG – each with its own (conceptual) security boundary. An entropy-source security boundary includes a noise source, health tests, and (optionally) a conditioning component. A DRBG security boundary contains the chosen DRBG, memory for the internal state, and health tests. An RBG security boundary contains health tests and may also contain an (optional) external conditioning function. The RBG2 and RBG3 constructions in Sections 5 and 6, respectively, use this model.

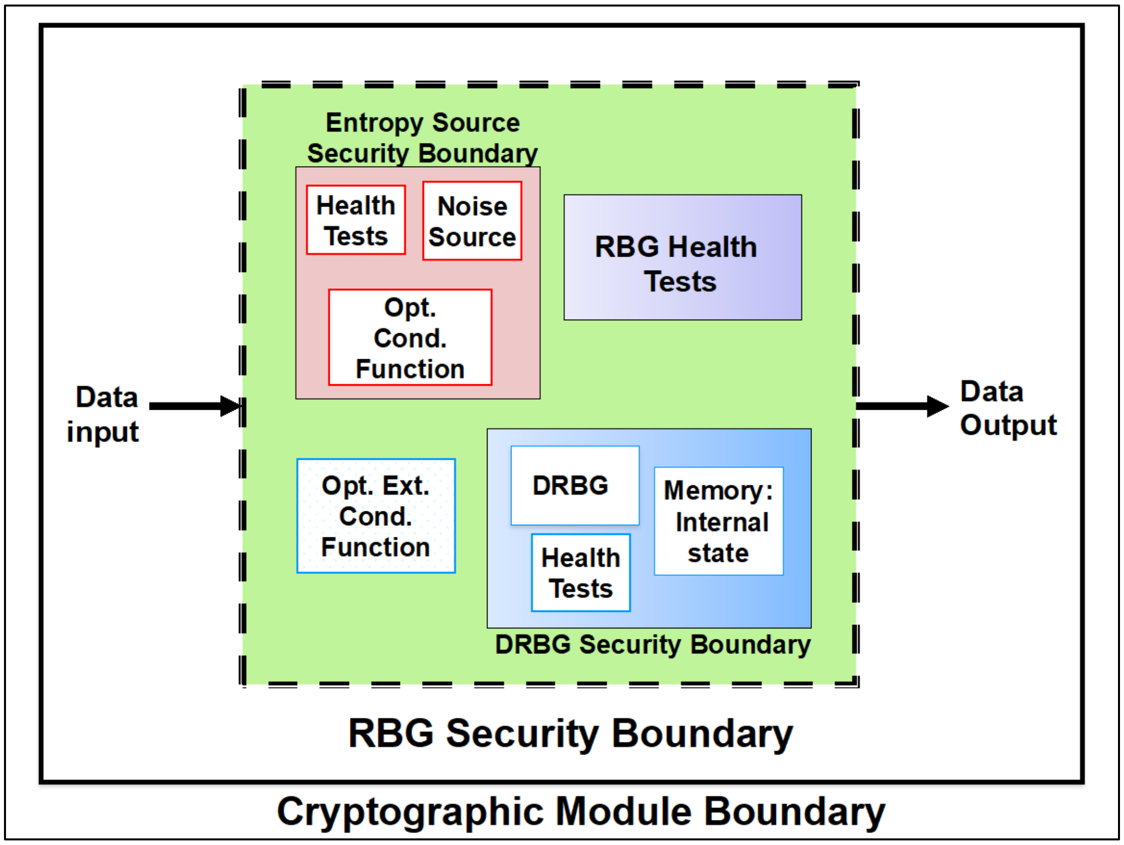

Fig. 2. Example of an RBG Security Boundary within a Cryptographic Module

Note that in the case of the RBG1 construction in Section 4, the security boundary containing the DRBG does not include a randomness source (shown as an entropy source in Figure 2).

A cryptographic primitive (e.g., an **approved** hash function) used by an RBG may be used by other applications within the same cryptographic module. However, these other applications **shall not** modify or reveal the RBG's output, intermediate values, or internal state.

2.6. Assumptions and Assertions

The RBG constructions in SP 800-90C are based on the use of validated entropy sources and the following assumptions and assertions for properly functioning entropy sources:

1. An entropy source is independent of another entropy source if a) their security boundaries do not overlap (e.g., they reside in separate cryptographic modules, or one is a physical entropy source and the other is a non-physical entropy source), b) there are no common noise sources,[4] and c) statistical tests provide evidence of the independence of the entropy sources.
2. The use of both validated and non-validated entropy sources is permitted in an implementation, but only entropy sources that have been validated for compliance with [SP800-90B] are used to provide the randomness input for seeding and reseeding a DRBG or providing entropy for an RBG3 construction.

The following assumptions and assertions pertain to the use of validated entropy sources for providing entropy bits:

3. For the purpose of analysis, it is assumed that a) the number of bits that are output by an entropy source is never more than 2^{64}, and b) the number of output bits from the RBG is never more than 2^{64} bits for a DRBG instantiation. In the case of an RBG1 construction with one or more subordinate DRBGs, the output limit applies to the total output provided by the RBG1 construction and all of its subordinate DRBGs.
4. Each entropy-source output has a fixed length, *ES_len* (in bits).
5. Each entropy-source output is assumed to contain a fixed amount of entropy, denoted as *ES_entropy*, that was assessed during entropy-source implementation validation. (See [SP800-90B] for entropy estimation.) *ES-entropy* is assumed to be at least 0.1 bits per bit of output.
6. Each entropy source has been characterized as either a physical entropy source or a non-physical entropy source upon successful validation.
7. The outputs from a single entropy source can be concatenated. The entropy of the resultant bitstring is the sum of the entropy from each entropy-source output. For example, if m outputs are concatenated, then the length of the bitstring is $m \times ES\_len$ bits, and the entropy for that bitstring is assumed to be $m \times ES\_entropy$ bits. (This is a consequence of the model of entropy used in [SP800-90B].)
8. The output of multiple independent entropy sources can be concatenated in an RBG. The entropy in the resultant bitstring is the sum of the entropy in the output of each independent entropy-source output that is considered to be contributing to the entropy in the bitstring (see Methods 1 and 2 in Section 2.3). For example, suppose that the output from independent physical entropy sources A and B and non-physical entropy source C are concatenated. The length of the concatenated bitstring is the sum of the lengths of the component bitstrings (i.e., $ES\_len_A + ES\_len_B + ES\_len_C$).

[4] They may, however, use the same *type* of noise source (e.g., both entropy sources could use ring oscillators but not the same ones).

- Using Method 1 in Section 2.3, the amount of entropy in the concatenated bitstring is $ES\_entropy_A + ES\_entropy_B$.
- Using Method 2 in Section 2.3, the amount of entropy in the concatenated bitstring is the sum of the entropies in the bitstrings (i.e., $ES\_entropy_A + ES\_entropy_B + ES\_entropy_C$).

9. Under certain conditions, the output of one or more entropy sources can be externally conditioned to provide full-entropy output. See Section 3.3.2 and Section 6.3.1 for the use of this assumption and [NISTIR8427] for rationale.

Furthermore,

10. The amount of entropy in a subset bitstring that is "extracted" from the output block of an approved hash function or block cipher is a proportion of the entropy in that block, such that

$$entropy_{subset} = \left(\frac{subset\_len}{output\_len}\right) entropy_{output\_block}$$

where *subset_len* is the length of the subset bitstring, *output_len* is the length of the output block, $entropy_{output\_block}$ is the amount of entropy in the output block, and $entropy_{subset}$ is the amount of entropy in the subset bitstring.

11. Full entropy bits can be extracted from the output block of a hash function or block cipher when the amount of fresh entropy inserted into the algorithm exceeds the number of bits to be extracted by at least 64 bits. For example, if *output_len* is the length of the output block, all bits of the output block can be assumed to have full entropy if at least *output_len* + 64 bits of entropy are inserted into the algorithm. As another example, if a DRBG is reseeded at its security strength *s*, (*s* – 64) bits with full entropy can be extracted from the DRBG's output block.

12. To instantiate a DRBG at a security strength of *s* bits, a bitstring of at least 3*s*/2 bits long is needed from a randomness source for an RBG1 construction, and a bitstring with at least 3*s*/2 bits of entropy is needed from an entropy source for an RBG2 or RBG3 construction.

13. One or more of the constructions provided herein are used in the design of an RBG.

14. All components of an RBG2 and RBG3 construction (as specified in Sections 5 and 6) reside within the physical boundary of a single [FIPS140]-validated cryptographic module.

15. The DRBGs specified in [SP800-90A] are assumed to meet their explicit security claims (e.g., backtracking resistance, prediction resistance, claimed security strength, etc.).

The following assumptions and assertions have been made for the subordinate DRBGs (sub-DRBGs) that are seeded (i.e., initialized) using an RBG1 construction:

16. A sub-DRBG is considered to be part of the RBG1 construction that initializes it.

17. The assumptions and assertions in items 3, 10, and 14 (above) apply to sub-DRBGs.

2.7. General Implementation and Use Requirements and Recommendations

When implementing the RBGs specified in this Recommendation, an implementation:

1. **Shall** destroy intermediate values before exiting the function or routine in which they are used,
2. **Shall** employ an "atomic" generate operation whereby a generate request is completed before using any of the requested bits,
3. **Should** consider the threats posed by quantum computers in the future, and
4. **Should** be implemented with the capability to support a security strength of 256 bits or to provide full-entropy output.

When using RBGs, the user or application requesting the generation of random or pseudorandom bits **should** request only the number of bits required for a specific immediate purpose rather than generating bits to be stored for future use. Since, in most cases, the bits are intended to be secret, the stored bits (if not properly protected) are potentially vulnerable to exposure, thus defeating the requirement for secrecy.

2.8. General Function Calls

Functions used within this document for accessing the DRBGs in [SP800-90A], the entropy sources in [SP800-90B], and the RBG3 constructions specified in SP 800-90C are provided below. Each function **shall** return a status code that **shall** be checked (e.g., a status of success or failure by the function).

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

If the status code indicates a failure of an RBG component, then see Section 7.1.2 for error-handling guidance. Note that if the status code does not indicate a success, an invalid output (e.g., a null bitstring) **shall** be returned with the status code if information other than the status code could be returned.

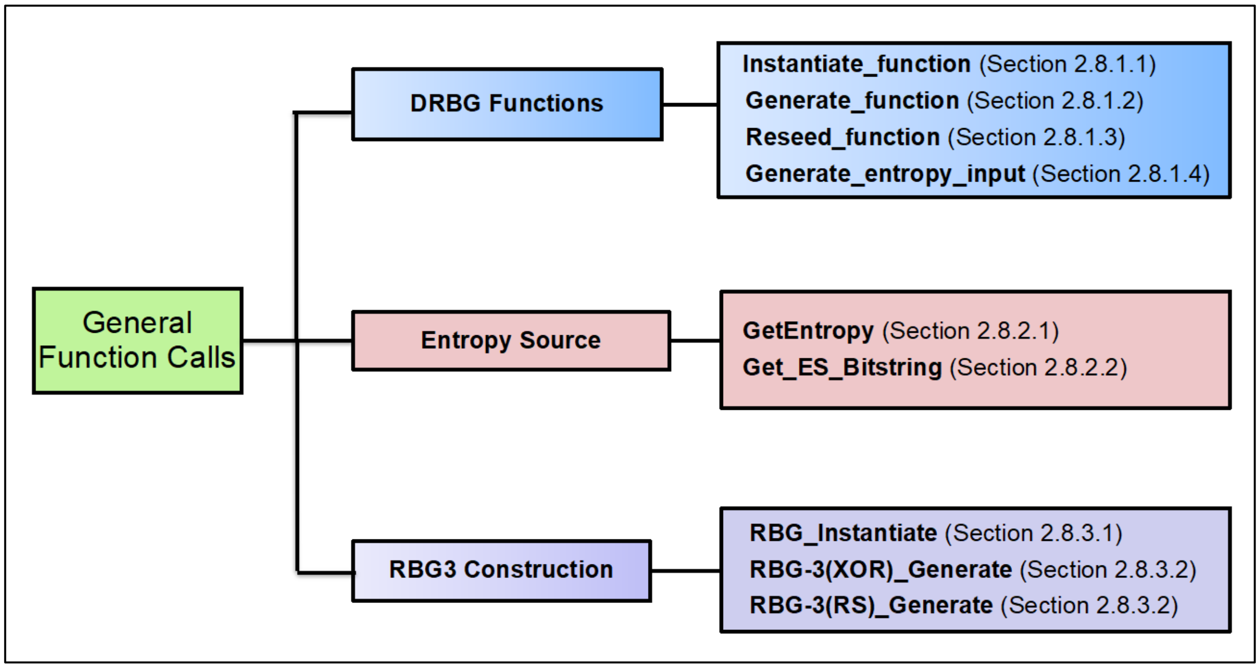
General Function Calls
DRBG Functions
Instantiate_function (Section 2.8.1.1)
Generate_function (Section 2.8.1.2)
Reseed_function (Section 2.8.1.3)
Generate_entropy_input (Section 2.8.1.4)
Entropy Source
GetEntropy (Section 2.8.2.1)
Get_ES_Bitstring (Section 2.8.2.2)
RBG3 Construction
RBG_Instantiate (Section 2.8.3.1)
RBG-3(XOR)_Generate (Section 2.8.3.2)
RBG-3(RS)_Generate (Section 2.8.3.2)

Fig. 3. General Function Calls

2.8.1. DRBG Functions

SP 800-90A specifies several functions for use within a DRBG, indicating the input and output parameters and other implementation details. Note that, in some cases, some input parameters may be omitted, and some output information may not be returned.

At least two functions are required in a DRBG:

1. An instantiate function that seeds the DRBG using the output of a randomness source and other input (see Section 2.8.1.1) and
2. A generate function that produces output for use by a consuming application (see Section 2.8.1.2).

A DRBG may also support a reseed function (see Section 2.8.1.3). A **Get_randomness-source_input** function is used in SP 800-90A to request output from a randomness source during instantiation and reseeding (see Section 2.8.1.4).

The use of the **Uninstantiate_function** specified in SP 800-90A is not explicitly discussed in SP 800-90C but may be required by an implementation.

2.8.1.1. DRBG Instantiation

A DRBG **shall** be instantiated prior to the generation of pseudorandom bits at the highest security strength to be supported by the DRBG instantiation using the following call:

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

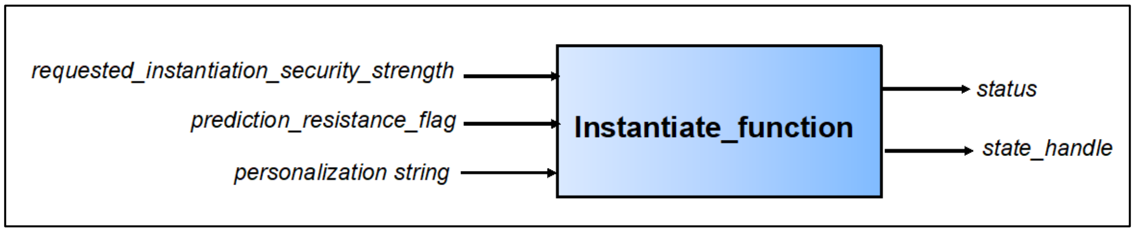
requested_instantiation_security_strength
prediction_resistance_flag
personalization string
Instantiate_function
status
state_handle

Fig. 4. Instantiate_function

The **Instantiate_function** (shown in Figure 4) is used to instantiate a DRBG at the *requested_instantiation_security_strength* using the output of a randomness source[5] and an optional *personalization_string* to create seed material. A *prediction_resistance flag* may be used to indicate whether subsequent **Generate_function** calls may request prediction resistance. As stated in Section 2.4.1, a *personalization_string* is optional but strongly recommended. (Details about the **Instantiate_function** are provided in [SP800-90A].)

If the returned status code for the **Instantiate_function** indicates a success (i.e., the DRBG has been instantiated at the requested security strength), a state handle may[6] be returned to indicate the particular DRBG instance. When provided, the state handle will be used in subsequent calls to the DRBG (e.g., during a **Generate_function** call) to identify the internal state information for the instantiation. The information in the internal state includes the security strength of the instantiation, the number of times that the instantiation has produced output, and other information that changes during DRBG execution (see [SP800-90A] for each DRBG design).

When the DRBG has been instantiated at the *requested_instantiation_security_strength*, the DRBG will operate at that security strength even if the *requested_security_strength* in subsequent **Generate_function** calls (see Section 2.8.1.2) is less than the instantiated security strength.

If the *status* code indicates an error and an implementation is designed to return a state handle, an invalid (e.g., *Null*) state handle **shall** be returned.

2.8.1.2. DRBG Generation Request

Pseudorandom bits are generated after DRBG instantiation using the following call:

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

[5] The randomness source provides the randomness input required to instantiate the security strength of the DRBG.
[6] In cases where only one instantiation of a DRBG will ever exist, a state handle need not be returned since only one internal state will be created.

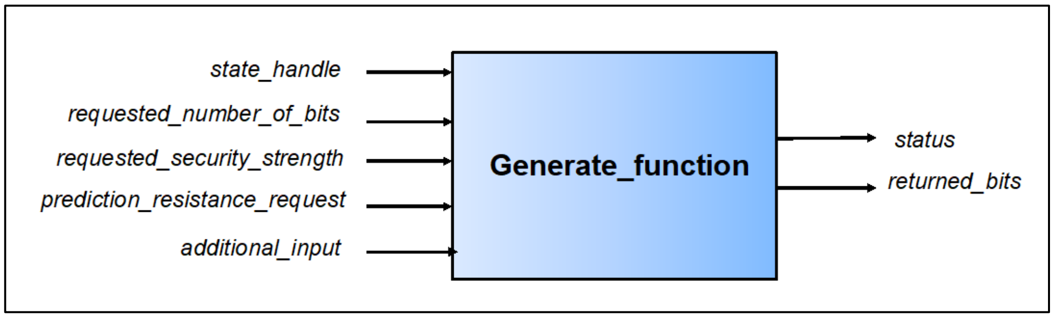

Fig. 5. Generate_function

The **Generate_function** (shown in Figure 5) requests that a DRBG generate a specified number of bits. The request may indicate the DRBG instance to be used (using the state handle returned by an **Instantiate_function** call; see Section 2.8.1.1), the number of bits to be returned, the security strength that the DRBG needs to support for generating the bitstring, and whether or not prediction resistance is to be obtained during this execution of the **Generate_function**. Optional additional input may also be incorporated into the function call. As stated in Section 2.4.1, the ability to handle and use additional input is recommended.

The **Generate_function** returns status information – either an indication of success or an error. If the returned *status* code indicates a success, the requested number of bits is returned.

- If *requested_number_of_bits* is equal to or greater than the instantiated security strength, the security strength that the *returned_bits* can support (if used as a key) is:

  ss_key = the instantiated security strength,

  where *ss_key* is the security strength of the key.

- If the *requested_number of bits* is less than the instantiated security strength, and the *returned_bits* are to be used as a key, the key is capable of supporting a security strength of:

  ss_key = *requested_number_of_bits*.

If the status code indicates an error, the *returned_bits* **shall** consist of an invalid (e.g., *Null)* bitstring that **must not** be used. Examples of conditions in which an error indication **shall** be returned include the following:

- The *requested_security_strength* exceeds the instantiated security strength for the DRBG (i.e., the security strength recorded in the DRBG's internal state during instantiation).
- Prediction resistance has been requested but cannot be obtained at this time.

Details about the **Generate_function** are provided in Section 9.3 of [SP800-90A].

2.8.1.3. DRBG Reseed Request

The reseeding of a DRBG instantiation is intended to insert additional entropy into that DRBG instantiation (e.g., to recover from a possible compromise or to provide prediction resistance). This is accomplished using the following call (note that this does not increase the security strength of the DRBG):

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

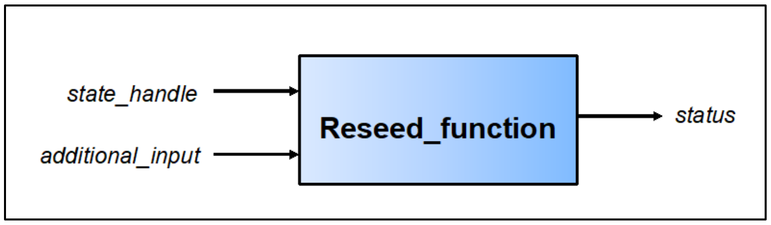

Fig. 6. Reseed_function

A **Reseed_function** (shown in Figure 6) is used to acquire at least *s* bits of fresh entropy for the DRBG instance indicated by the state handle (or the only instance if no state handle has been provided), where *s* is the security strength of the DRBG.[7] In addition to the randomness input provided from the randomness source(s) during reseeding, optional additional input may be incorporated into the reseed process. As discussed in Section 2.4.1, the capability for handling and using additional input is recommended. (Details about the **Reseed_function** are provided in [SP800-90A].)

An indication of the *status* is returned.

The **Reseed_function** is not permitted in an RBG1 construction (see Section 4) but is permitted in the RBG2 and RBG3 constructions (see Sections 5 and 6, respectively).

2.8.1.4. The Get_randomness-source_input Call

A **Get_randomness-source_input** call is used in the **Instantiate_function** and **Reseed_function** in [SP800-90A] to indicate when a randomness source (i.e., an entropy source or RBG) needs to be accessed to obtain randomness input. Details are not provided in SP 800-90A about how the **Get_randomness-source_input** call needs to be implemented. SP 800-90C provides guidance on how the call should actually be implemented based on various situations. Sections 4, 5, and 6 provide instructions for obtaining input from a randomness source when the **Get_randomness-source_input** call is encountered in SP 800-90A.[8]

2.8.2. Interfacing with Entropy Sources Using the GetEntropy and Get_ES_Bitstring Functions

2.8.2.1. The GetEntropy Call

An entropy source, as discussed in [SP800-90B], is a mechanism for producing bitstrings that cannot be predicted and whose unpredictability can be quantified in terms of min-entropy. SP 800-90B uses the following call for accessing an entropy source:

(*status*, *ES_output*) = **GetEntropy** (*bits_of_entropy*),

[7] The value of *s* is available in the DRBG's internal state.
[8] Note that, at this time, modifications to the **Instantiate_function** and **Reseed_function** specification in SP 800-90A and to the appropriate algorithms in Section 10 of that document may be required to accommodate the specific requests for entropy for each RBG construction.

where *bits_of_entropy* is the amount of entropy requested, *ES_output* is a bitstring containing the requested amount of entropy, and *status* indicates whether or not the request has been satisfied. See Figure 7.

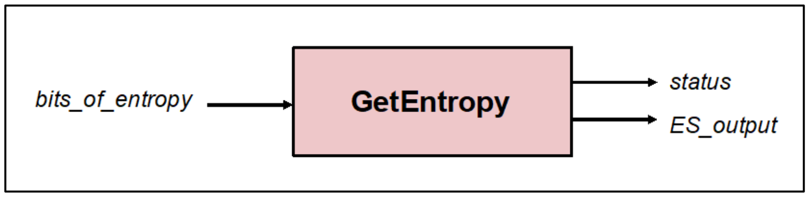

Fig. 7. GetEntropy function

If the *status* indicates a success, a bitstring of at least *bits_of_entropy* long is returned as the *ES_output*. *ES_output* **must** contain at least the requested amount of entropy indicated by the *bits_of_entropy* input parameter. If the *status* does not indicate a success, an invalid *ES_output* bitstring is returned (e.g., *ES_output* could be a null bitstring).

2.8.2.2. The Get_ES_Bitstring Function

A single **GetEntropy** call may not be sufficient to obtain the entropy required for seeding and reseeding a DRBG and for providing input for the exclusive-or operation in an RBG3(XOR) construction (see Section 6.2). Therefore, SP 800-90C uses a **Get_ES_Bitstring** function (see Figure 8) to obtain the required entropy from one or more **GetEntropy** calls. The **Get_ES_Bitstring** function is invoked as follows:

(*status*, *entropy_bitstring*) = **Get_ES_Bitstring**(*bits_of_entropy*),

where *bits_of_entropy* is the amount of entropy requested in the returned *entropy_ bitstring*, and *status* indicates whether or not the request has been satisfied.

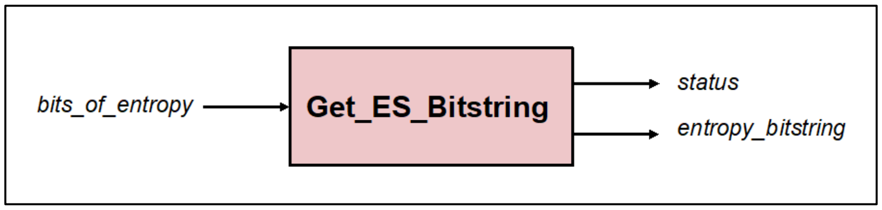

Fig. 8. Get_ES_Bitstring function

Note that if non-validated entropy sources are used (e.g., to provide entropy to be used as additional input), they **shall** be accessed using a different function than is used to access validated entropy sources (i.e., the **Get_ES_Bitstring** function).

If the returned *status* from the **Get_ES_Bitstring** function indicates a success, the requested amount of entropy (i.e., indicated by *bits_of_entropy*) **shall** be returned in the *entropy_bitstring*, whose length is equal to or greater than *bits_of_entropy*. If the *status* does not indicate a success, an invalid *entropy_bitstring* **shall** be returned (e.g., *entropy_bitstring* is a null bitstring).

The **Get_ES_Bitstring** function will be used in this document to access validated entropy sources to obtain one or more bitstrings with entropy using **GetEntropy** calls.

See Section 3.1 for additional discussion about the **Get_ES_Bitstring** function.

2.8.3. Interfacing with an RBG3 Construction

An RBG3 construction requires interface functions to instantiate its DRBG (see Section 2.8.3.1) and to request the generation of full-entropy bits (see Section 2.8.3.2).

2.8.3.1. Instantiating a DRBG within an RBG3 Construction

The **RBG3_DRBG_Instantiate** function is used to instantiate the DRBG within the RBG3 construction using the following call:

(*status*, *state_handle*) = **RBG3_DRBG_Instantiate**(*prediction_resistance_flag*, *personalization_string*).

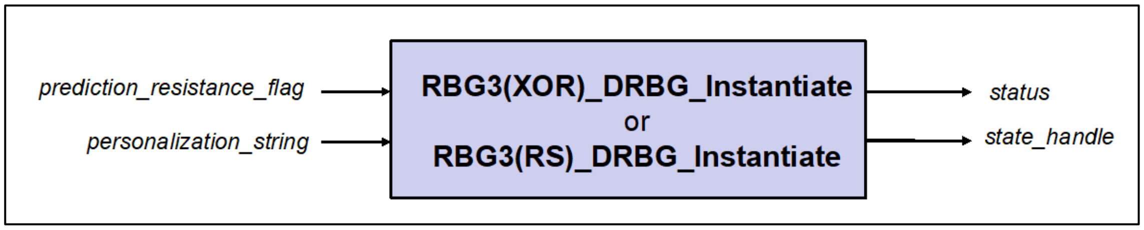
prediction_resistance_flag
personalization_string
RBG3(XOR)_DRBG_Instantiate
or
RBG3(RS)_DRBG_Instantiate
status
state_handle

Fig. 9. RBG3 DRBG_Instantiate function

The RBG3's instantiate function (shown in Figure 9) will result in a call to the DRBG's **Instantiate_function** (provided in Section 2.8.1.1). An optional but recommended *personalization_string* (see Section 2.4.1) may be provided as an input parameter. If included, the *personalization_string* **shall** be passed to the DRBG that is instantiated in the **Instantiate_function** request. See Sections 6.2.1.1 and 6.3.1.1 for more specificity.

If the returned *status* code indicates a success, a state handle may be returned to indicate the particular DRBG instance that is to be used by the construction. Note that if multiple instances of the DRBG are used, a separate state handle **shall** be returned for each instance. When provided, the state handle **shall** be used in subsequent calls to that RBG (e.g., during a call to the generate function) when multiple instances of the DRBG have been instantiated. If the status code indicates an error (e.g., entropy is not currently available, or the entropy source has failed), an invalid (e.g., *Null*) state handle **shall** be returned.

2.8.3.2. Generation Using an RBG3 Construction

The RBG3(XOR) and RBG3(RS) generate functions are different because of the difference in their designs (see Sections 6.2.1.2 and 6.3.1.2).

For the RBG3(XOR) construction, the generate function is invoked using the following call:

(*status*, *returned_bits*) = **RBG3(XOR)_Generate**(*state_handle*, *requested_number_of_bits, prediction_resistance_request, additional_input*).

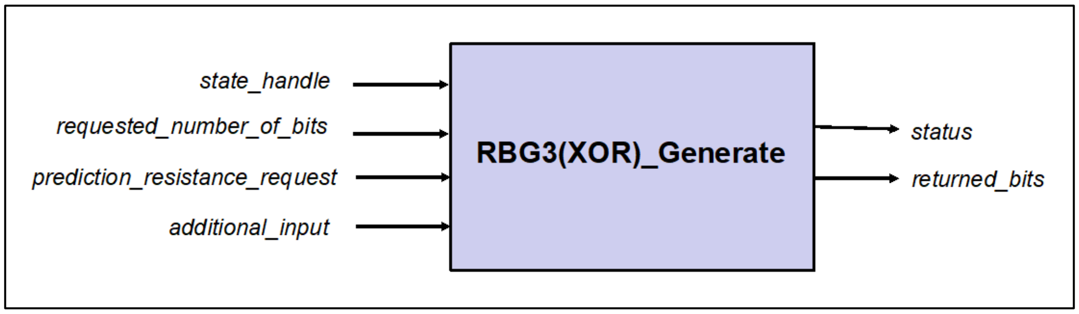

Fig. 10. RBG3(XOR)_Generate function

For the RBG3(RS) construction, the generate function is invoked using the following call:

(*status*, *returned_bits*) = **RBG3(RS)_Generate**(*state_handle*, *requested_number_of_bits, additional_input*).

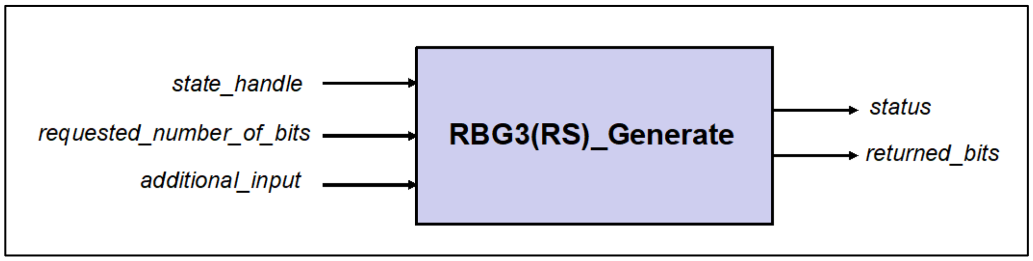

Fig. 11. RBG3(RS)_Generate function

The **RBG3(XOR)_Generate** function (shown in Figure 10) includes a *prediction_resistance_request* parameter to request a reseed of the RBG3(XOR)'s DRBG instantiation, when desired. This parameter is not included as a parameter for the **RBG3(RS)_Generate** function (shown in Figure 11) since this design always reseeds itself during execution.

The generate functions result in calls to the entropy sources and the DRBG instantiation used by the RBG3 construction. This call accesses the DRBG using the **Generate_function** call provided in Section 2.8.1.2. The input parameters to the two generate functions are used when calling the DRBG instantiation used by that RBG3 construction.

If the returned status code indicates a success, a bitstring that contains the newly generated bits is returned. The RBG then uses the resulting bitstring as specified for each RBG3 construction (see Section 6).

If the status code indicates an error (e.g., the entropy source has failed), an invalid (e.g., *Null*) bitstring **shall** be returned as the *returned_bits*.

3. Accessing Entropy Source Output

The security provided by an RBG is based on the use of validated entropy sources. Section 3.1 discusses the use of the **Get_ES_Bitstring** function to request entropy from one or more entropy sources. Section 3.2 discusses the behavior required by an entropy source. Section 3.3 discusses the conditioning of the output of one or more entropy sources to obtain a bitstring with full entropy before further use by an RBG.

3.1. The Get_ES_Bitstring Function

The **Get_ES_Bitstring** function specified in Section 2.8.2.2 is used within an RBG to obtain entropy from one or more validated entropy sources using one or more **GetEntropy** calls (see Sections 2.8.2.1 and 3.2) in whatever manner is required (e.g., by polling the entropy sources or by extracting bits containing entropy from a pool of collected bits). The **Get_ES_Bitstring** function **shall** only be used to access validated entropy sources to obtain the entropy for seeding and reseeding a DRBG and for providing input for the exclusive-or operation of an RBG3(XOR) construction (see Section 6.2).

In many cases, the **Get_ES_Bitstring** function will need to query an entropy source (or a set of entropy sources) multiple times to obtain the amount of entropy requested. For the most part, the construction of the **Get_ES_Bitstring** function itself is not specified in this document but is left to the developer to implement appropriately for the selected entropy sources.

The behavior of the **Get_ES_Bitstring** function **shall** be as follows:

1. A **Get_ES_Bitstring** function **shall** only be used to access one or more validated entropy sources.
2. The entropy bitstrings produced from multiple entropy-source calls to a single validated entropy source or by calls to multiple validated entropy sources **shall** be concatenated into a single bitstring. The entropy in the bitstring is computed as the sum of the entropy produced by each call to a validated entropy source that is to be counted as contributing entropy to the bitstring (see Section 2.3).[9]
3. If a failure is reported during an invocation of the **Get_ES_Bitstring** function by any physical or non-physical entropy source whose entropy is counted toward fulfilling an entropy request, the failure **shall** be handled as discussed in Section 7.1.2.
4. If a non-physical entropy source whose entropy is not counted reports a failure, the failure **shall** be reported to the RBG or the consuming application.
5. The **Get_ES_Bitstring** function **shall** not return an *entropy_bitstring* unless the bitstring contains sufficient entropy to fulfill the entropy request. The returned *status* **shall** indicate a success only when this condition is met.

[9] For Method 1 in Section 3.3, only entropy contributed by one or more validated physical entropy sources is counted. For Method 2, the entropy from all validated entropy sources is counted.

3.2. Entropy Source Requirements

This Recommendation requires the use of one or more validated entropy sources to provide entropy for seeding and reseeding a DRBG and for input to the XOR operation in the RBG3(XOR) construction specified in Section 6.2. In addition to the assumptions and assertions concerning entropy sources in Section 2.6, the following conditions **shall** be met when using these entropy sources:

1. Only validated entropy sources **shall** be used to provide the entropy bitstring for seeding and reseeding a DRBG and for providing input to the XOR operation in the RBG3(XOR) construction.

   Non-validated entropy sources may be used by an RBG to provide input for personalization strings and/or the additional input in DRBG function calls (see Section 2.4.1).

2. Each validated entropy source **shall** be independent of all other validated or non-validated entropy sources used by the RBG.
3. The outputs from an entropy source **shall not** be reused (e.g., the value in the entropy source is erased after being output).
4. When queried for entropy, the validated entropy sources **must** respond as follows:
   a. The requested output **must** be returned only if the returned status indicates a success. In this case, the *ES-output* bitstring **must** contain the requested amount of entropy. (Note that the *ES-output* bitstring may be longer than the amount of entropy requested, i.e., the bitstring may not have full entropy.)
   b. If an indication of a failure is returned by a validated entropy source as the status, an invalid (e.g., *Null*) bitstring **shall** be returned as *ES_output*.
5. If the validated entropy-source components operate continuously regardless of whether requests are received and a failure is determined, the entropy source **shall** immediately report the failure to the RBG (see Section 7.1.2).
6. If a validated entropy source reports a failure (e.g., because of a failed health test), the entropy source **shall not** produce output (except possibly for a failure status indication) until the failure is corrected. The entropy source **shall** immediately report the failure to the **Get_ES_Bitstring** function (see Section 3.1). If multiple validated entropy sources are used, the report **shall** identify the entropy source that reported the failure.
7. A detected failure of any entropy source **shall** cause the RBG to report the failure to the consuming application and terminate the RBG operation. The RBG **must not** be returned to normal operation until the conditions that caused the failure have been corrected and tested for successful operation.

3.3. External Conditioning to Obtain Full-Entropy Bitstrings

An RBG3(XOR) construction (see Section 6.2) and a CTR_DRBG without a derivation function in an RBG2 or RBG3 construction (see Sections 5 and 6) require bitstrings with full entropy from an entropy source. If the validated entropy source does not provide full-entropy output, a method for conditioning the output to obtain a bitstring with full entropy is needed. Since this conditioning is performed outside an entropy source, the output is said to be *externally conditioned*.

When external conditioning is performed, the vetted conditioning function listed in [SP800-90B] **shall** be used.

3.3.1. Conditioning Function Calls

The conditioning functions operate on bitstrings obtained from one or more calls to the entropy source(s).

The following format is used in Section 3.3.2 for a conditioning-function call:

conditioned_output = **Conditioning_function**(*input_parameters*),

where the *input_parameters* for the selected conditioning function are discussed in Sections 3.3.1.2 and 3.3.1.3, and *conditioned_output* is the output returned by the conditioning function.

3.3.1.1. Keys Used in External Conditioning Functions

The **HMAC**, **CMAC**, and **CBC-MAC** vetted conditioning functions require the input of a *Key* of a specific length (*keylen)*. Unlike other cryptographic applications, keys used in these external conditioning functions do not require secrecy to accomplish their purpose so may be hard-coded, fixed, or all zeros.

For the **CMAC** and **CBC-MAC** conditioning functions, the length of the key **shall** be an **approved** key length for the block cipher used (e.g., *keylen* = 128, 192, or 256 bits for AES).

For the **HMAC** conditioning function, the length of the key **shall** be equal to the length of the hash function's output block (i.e., *output_len*).

Table 2. Key Lengths for the Hash-based Conditioning Functions

Hash Function	**Length of the output block (*output_len*) and key (*keylen*)**
SHA-224, SHA-512/224, SHA3-224	224
SHA-256, SHA-512/256, SHA3-256	256
SHA-384, SHA3-384	384
SHA-512, SHA3-512	512

Using random keys may provide some additional security in case the input is more predictable than expected. Thus, these keys **should** be chosen randomly in some way (e.g., by drawing bits directly from the entropy source and inserting them into the key or by providing entropy-source bits to a conditioning function with a fixed key to derive the new key). Note that any entropy used to randomize the key **shall not** be used for any other purpose (e.g., as input to the conditioning function).

3.3.1.2. Hash Function-based Conditioning Functions

Conditioning functions may be based on **approved** hash functions.

One of the following calls **shall** be used for external conditioning when the conditioning function is based on a hash function:

1. Using an **approved** hash function directly:

   conditioned_output = **Hash**(*entropy_bitstring*),

   where the hash function operates on the *entropy_bitstring* provided as input.

2. Using HMAC with an **approved** hash function:

   conditioned_output = **HMAC**(*Key*, *entropy_bitstring*),

   where HMAC operates on the *entropy_bitstring* using a *Key* determined as specified in Section 3.3.1.1.

3. Using Hash_df as specified in SP 800-90A:

   conditioned_output = **Hash_df**(*entropy_bitstring*, *output_len*),

   where the derivation function operates on the *entropy_bitstring* provided as input to produce a bitstring of *output_len* bits.

In all three cases, the length of the conditioned output is equal to the length of the output block of the selected hash function (i.e., *output_len*).

3.3.1.3. Block Cipher-based Conditioning Functions

Conditioning functions may be based on **approved** block ciphers.[10] TDEA **shall not** be used as the block cipher (see Section 2.6).

For block cipher-based conditioning functions, one of the following calls **shall** be used for external conditioning:

1. Using CMAC (as specified in [SP800-38B]) with an **approved** block cipher:

   conditioned_output = **CMAC**(*Key*, *entropy_bitstring*),

   where CMAC operates on the *entropy_bitstring* using a *Key* determined as specified in Section 3.3.1.1.

2. Using CBC-MAC (specified in Appendix F of [SP800-90B]) with an **approved** block cipher:

   conditioned_output = **CBC-MAC**(*Key*, *entropy_bitstring*),

   where CBC-MAC operates on the *entropy_bitstring* using a *Key* determined as specified in Section 3.3.1.1.

[10] At the time of publication, only AES-128, AES-192, and AES-256 were **approved** as block ciphers for the conditioning functions (see SP 800-90B). In all three cases, the block length is 128 bits.

CBC-MAC **shall** only be used as an external conditioning function under the following conditions:

a. The length of the input is an integer multiple of the block size of the block cipher (e.g., a multiple of 128 bits for AES) – no padding is done by CBC-MAC itself.[11]

b. All inputs to CBC-MAC in the same RBG **shall** have the same length.

c. If the CBC-MAC conditioning function is used to obtain full entropy from an entropy source for CTR_DRBG instantiation or reseeding:

- A personalization string **shall not** be used during instantiation.
- Additional input **shall not** be used during the reseeding of the CTR_DRBG but may be used during the generate process.

CBC-MAC is not approved for any use other than in an RBG (see [SP800-90B]).

3. Using the **Block_cipher_df** as specified in [SP800-90A] with an **approved** block cipher:

conditioned_output = **Block_cipher_df**(*entropy_bitstring*, *block_length*),

where **Block_cipher_df** operates on the *entropy_bitstring* using a key specified within the function, and the *block_length* is 128 bits for AES.

In all three cases, the length of the conditioned output is equal to the length of the output block (i.e., 128 bits for AES). If the requested amount of entropy is requested for subsequent use by an RBG,[12] then multiple iterations of the conditioning function may be required, each using a different *entropy_bitstring*.

3.3.2. Using a Vetted Conditioning Function to Obtain Full-Entropy Bitstrings

This construction will produce a bitstring with full entropy using one of the conditioning functions identified in Section 3.3.1.1 for an RBG2 or RBG3 construction whenever a bitstring with full entropy is required (e.g., to seed or reseed a CTR_DRBG with no derivation function or to provide full entropy for the RBG3(XOR) construction). This process is unnecessary if the entropy source provides full-entropy output.

Let *output_len* be the length of the output block of the vetted conditioning function to be used; *output_len* is the length of the hash function's output block when a hash-based conditioning function is used (see Section 3.3.1.2); *output_len* = 128 when an AES-based conditioning function is used (see Section 3.3.1.3).

The approach used by this construction is to acquire sufficient entropy from the entropy source to produce *output_len* bits with full entropy in the conditioning function's output block, where *output_len* is the length of the output block. The amount of entropy required for each use of the conditioning function is *output_len* + 64 bits (see item 11 of Section 2.6). This process is repeated until the requested number of full-entropy bits have been produced.

[11] Any padding required could be done before submitting the *entropy_bitstring* to the CBC-MAC function.
[12] Since the output block of AES is only 128 bits, this will often be the case when seeding or reseeding a DRBG.

The **Get_conditioned_full_entropy_ input** function below obtains entropy from one or more entropy sources using the **Get_ES_Bitstring** function discussed in Section 3.1 and conditions it to provide an *n*-bit string with full entropy.

```
Get_conditioned_full_entropy_input:
    Input: integer $n$.                    Comment: the requested number of full-entropy bits.
    Output: integer $status$, bitstring $returned\_bitstring$.
Process:
    1. $temp$ = the $Null$ string.
    2. $ctr = 0$.
    3. While $ctr < n$, do
        3.1 ($status$, $entropy\_bitstring$) = Get_ES_Bitstring($output\_len + 64$).
        3.2 If ($status \neq$ SUCCESS), then return ($status$, $invalid\_bitstring$).
        3.3 $conditioned\_output$ = Conditioning_function($input\_parameters$).
        3.4 $temp = temp \,||\, conditioned\_output$.
        3.5 $ctr = ctr + output\_len$.
    4. $returned\_bitstring$ = leftmost($temp$, $n$).
    5. Return (SUCCESS, $returned\_bitstring$).
```

Steps 1 and 2 initialize the temporary bitstring (*temp*) for storing the full-entropy bitstring being assembled and the counter (*ctr*) that counts the number of full-entropy bits produced for each iteration of step 3.

Step 3 obtains and processes the entropy for each iteration.

- Step 3.1 requests *output_len* + 64 bits from the validated entropy sources. When the output of multiple entropy sources is used, the entropy counted for fulfilling the request for *outlen* + 64 bits is determined using Method 1 or Method 2 as specified in Section 2.3 in the following situations:

  Method 1 **shall** be used when:

  Instantiating and reseeding an RBG2(P) construction containing a CTR_DRBG with no derivation function (see Section 5.2.1, item 1b, and Section 5.2.3),

  Instantiating and reseeding a CTR_DRBG with no derivation function that is used within an RBG3 construction (see Section 6.1, requirement 1), or

  Generating bits in an RBG3(XOR) construction (see Section 6.2.1.2, step 1).

  Method 2 **shall** be used when instantiating and reseeding an RBG2(NP) construction containing a CTR_DRBG with no derivation function (see Section 5.2.1, item 1b, and Section 5.2.3).

- Step 3.2 checks whether or not the *status* returned in step 3.1 indicated a success. If the *status* did not indicate a success, the *status* is returned along with an invalid bitstring as the *returned_bitstring* (e.g., *invalid_bitstring* is *Null*).
- Step 3.3 invokes the conditioning function for processing the *entropy_bitstring* obtained from step 3.1. The *input_parameters* for the selected **Conditioning_function** are specified in Sections 3.3.1.2 or 3.3.1.3, depending on the conditioning function used.
- Step 3.4 concatenates the *conditioned_output* received in step 3.3 to the temporary bitstring (*temp*), and step 3.5 increments the counter for the number of full-entropy bits that have been produced so far.
- If at least *n* full-entropy bits have not been produced, repeat the process starting at step 3.1.
- Step 4 truncates the full-entropy bitstring to *n* bits.
- Step 5 returns an *n*-bit full-entropy bitstring as the *returned_bitstring*.

4. RBG1 Constructions Based on RBGs with Physical Entropy Sources

An RBG1 construction provides a source of cryptographic random bits from a device that has no internal randomness source. Its security depends entirely on being instantiated securely from an RBG with access to a physical entropy source that resides outside of the device.

An RBG1 construction is instantiated (i.e., seeded) only once before its first use by an RBG2(P) construction (see Section 5) or an RBG3 construction (see Section 6). Since a randomness source is not available after DRBG instantiation, an RBG1 construction cannot be reseeded and, therefore, cannot provide prediction resistance.

An RBG1 construction may be useful for constrained devices in which an entropy source cannot be implemented or in any device in which access to a suitable source of randomness is not available after instantiation. Since an RBG1 construction cannot be reseeded, the use of the DRBG is limited to the DRBG's seedlife (see [SP800-90A]).

Subordinate DRBGs (sub-DRBGs) may be used within the security boundary of an RBG1 construction (see Section 4.3). The use of one or more sub-DRBGs may be useful for implementations that use flash memory, such as when the number of write operations to the memory is limited (resulting in short device lifetimes) or when there is a need to use different DRBG instantiations for different purposes. The RBG1 construction is the source of the randomness that is used to (optionally) instantiate one or more sub-DRBGs. Each sub-DRBG is a DRBG specified in SP 800-90A and is intended to be used for a limited time and a limited purpose. A sub-DRBG is, in fact, a different instantiation of the DRBG design implemented within the RBG1 construction (see Section 2.4.1).

4.1. RBG1 Description

As shown in Figure 12, an RBG1 construction consists of a DRBG contained within a DRBG security boundary in one cryptographic module and an RBG (serving as a randomness source) contained within a separate cryptographic module from that of the RBG1 construction. Note that the required health tests are not shown in the figure.

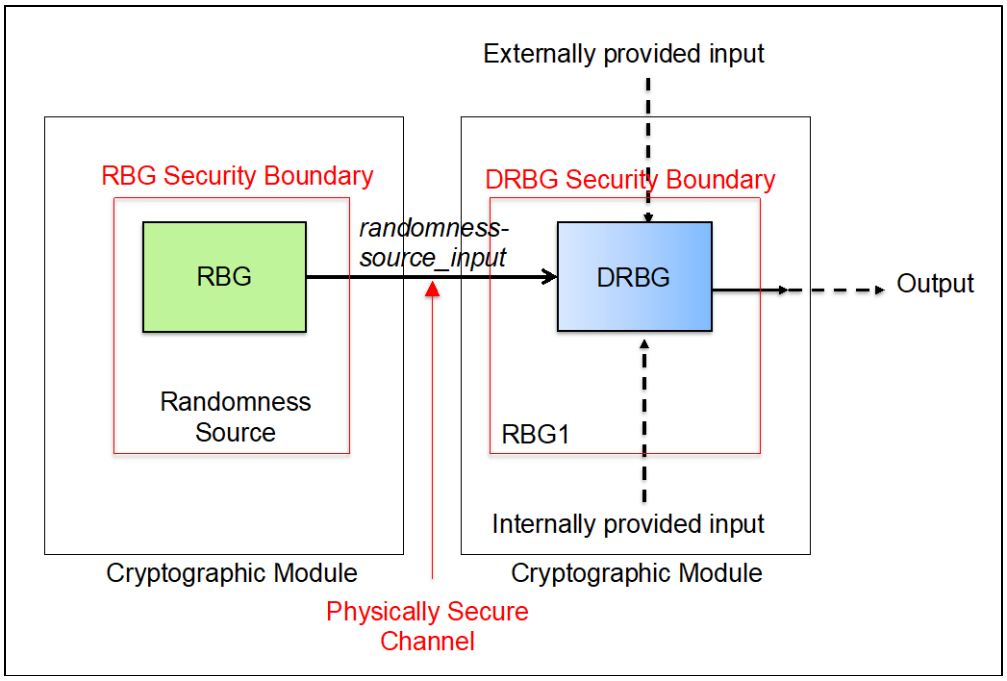

Fig. 12. RBG1 Construction

The RBG for instantiating the DRBG within the RBG1 construction **must** be either an RBG2(P) construction that has support for prediction resistance requests (see Section 5) or an RBG3 construction (see Section 6). A physically secure channel between the randomness source and the DRBG is used to securely transport the randomness input required for the instantiation of the DRBG. An optional recommended personalization string and optional additional input may be provided from within the DRBG's cryptographic module or from outside of that module (see Section 2.4.1).

An external conditioning function is not needed for this design because the output of the RBG has already been cryptographically processed.

The output from an RBG1 construction may be used within the cryptographic module (e.g., to seed a sub-DRBG as specified in Section 4.3) or by an application outside of the RBG1 security boundary.

The security strength provided by the RBG1 construction is the minimum of the security strengths provided by the DRBG within the construction, the secure channel, and the RBG used to seed the DRBG.

Examples of RBG1 and sub-DRBG constructions are provided in Appendices B.2 and B.3, respectively.

4.2. Conceptual Interfaces

Interfaces to the DRBG within an RBG1 construction include function calls for instantiating the DRBG and generating pseudorandom bits upon request (see Sections 4.2.1 and 4.2.2).

Note that reseeding is not included in this construction.

4.2.1. Instantiating the DRBG in the RBG1 Construction

The DRBG within the RBG1 construction may be instantiated at any security strength possible for the DRBG design using the **Instantiate_function** discussed in Section 2.8.1.1 and [SP800-90A], subject to the maximum security strength that is supported by the RBG used as the randomness source.

(*status*, *RBG1_state_handle*) =
Instantiate_function (*s, prediction_resistance_flag* = FALSE*, personalization_string*),

where *s* is the requested security strength for the DRBG in the RBG1 construction. If used, the *prediction_resistance_flag* is set to FALSE since the DRBG cannot be reseeded to provide prediction resistance.

An external RBG (i.e., the randomness source) **shall** be used to obtain the bitstring necessary for establishing the DRBG's *s*-bit security strength.

In SP 800-90A, the **Instantiate_function** specifies the use of a **Get_randomness-source_input** call to obtain randomness input from the randomness source for instantiation (see Section 2.8.1.4 in this document and in [SP800-90A]). For an RBG1 construction, an **approved** external RBG2(P) or RBG3 construction **must** be used as the randomness source (see Sections 5 and 6, respectively).

If the randomness source is an RBG2(P) construction (see Figure 13), the **Get_randomness-source_input** call in the **Instantiate_function shall** be replaced by a **Generate_function** call to the RBG2(P) construction (in whatever manner is required) (see Sections 2.8.1.2 and 5.2.2). The RBG2(P) construction **must** be reseeded using its internal entropy source(s) before generating bits to be provided to the RBG1 construction. This is accomplished by setting the *prediction_resistance_request* parameter in the **Generate_function** call to TRUE (see steps 1a and 2a below).

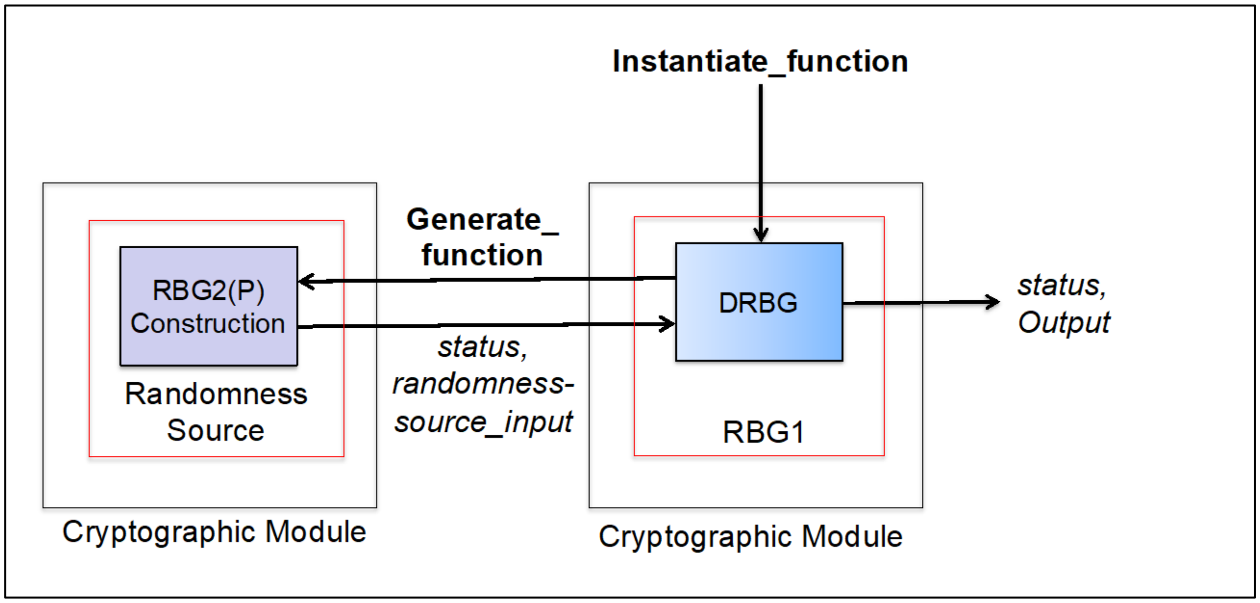

Fig. 13. Instantiation Using an RBG2(P) Construction as a Randomness Source

If the randomness source is an RBG3 construction (as shown in Figure 14), the **Get_randomness-source_input** call **shall** be replaced by the appropriate RBG3 generate function (see Sections 2.8.3.2, 6.2.1.2, and 6.3.1.2 and steps 1b, 1c, 2b, and 2c below).

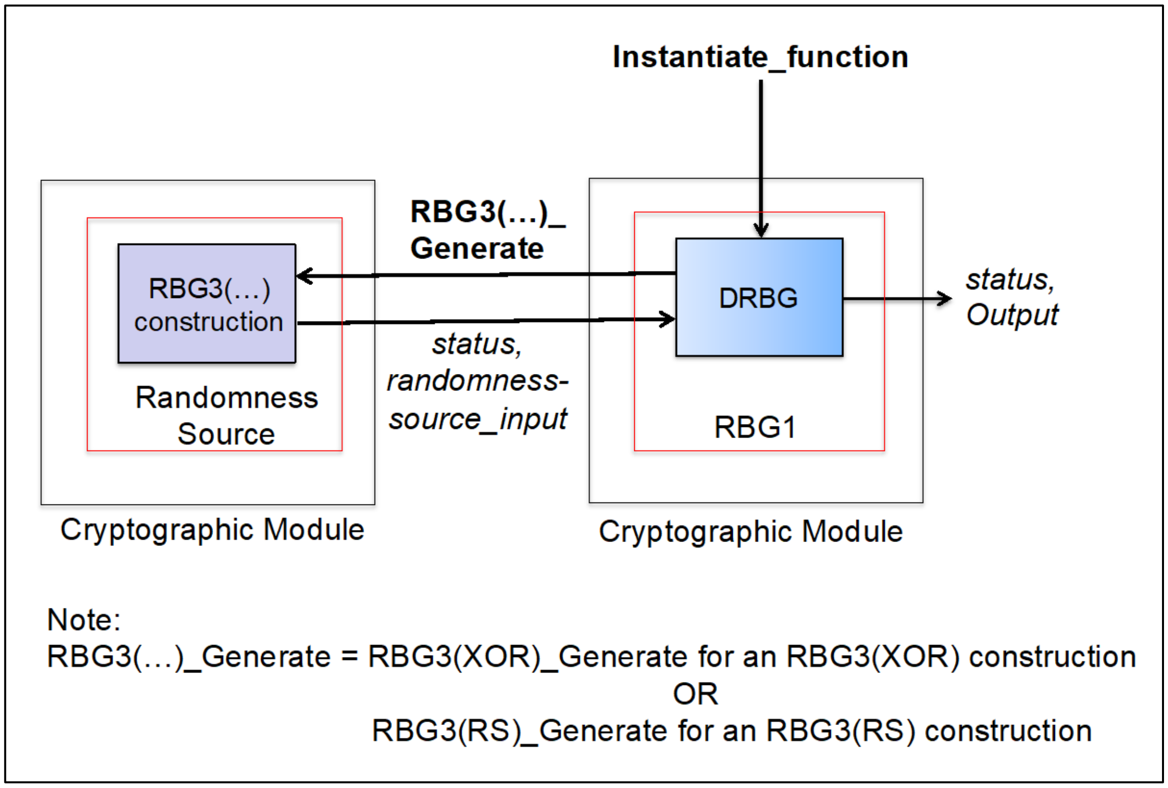

Fig. 14. Instantiation using an RBG3(XOR) or RBG3(RS) Construction as a Randomness Source

Let *s* be the security strength to be instantiated. The DRBG within an RBG1 construction is instantiated as follows:

1. When an RBG1 construction is instantiating a CTR_DRBG without a derivation function, $s + 128$ bits[13] **shall** be obtained from the randomness source as follows:

   If the randomness source is an RBG2(P) construction (see Figure 13), the **Get_randomness-source_input** call is replaced by:

   (*status*, *randomness-source_input*) = **Generate_function**(*RBG2_state_handle*, s + 128, *s*, *prediction_resistance_request* = TRUE, *additional_input*).

   Note that the DRBG within the RBG2(P) construction **must** be reseeded before generating output.[14] This may be accomplished by requesting prediction resistance (i.e., setting *prediction_resistance_request* = TRUE). See Requirement 17 in Section 4.4.1.

[13] For AES, the block length is 128 bits, and the key length is equal to the security strength *s*. SP 800-90A requires the randomness input from the randomness source to be key length + block length bits when a derivation function is not used.
[14] See Requirement 11 in Section 5.4.1.

If the randomness source is an RBG3(XOR) construction (see Figure 14), the **Get_randomness-source_input** call is replaced by:

(*status*, *randomness-source_input*) = **RBG3(XOR)_Generate**(*RBG3_state_handle*, *s* + 128, *prediction_resistance_request*, *additional_input*).

A request for prediction resistance from the DRBG used by the RBG3(XOR) construction is optional.

c) If the randomness source is an RBG3(RS) construction (see Figure 14), the **Get_randomness-source_input** call is replaced by:

(*status*, *randomness-source_input*) = **RBG3(RS)_Generate**(*RBG3_state_handle*, 3*s*/2, *additional_input*).

2. When an RBG1 construction is instantiating any other DRBG (including a CTR_DRBG with a derivation function), 3*s*/2 bits **shall** be obtained from a randomness source that provides a security strength of at least *s* bits.

   a) If the randomness source is an RBG2(P) construction (see Figure 13), the **Get_randomness-source_input** call is replaced by:

   (*status*, *randomness-source_input*) = **Generate_function**(*RGB2_state_handle*, 3*s*/2, *s*, *prediction_resistance_request* = TRUE, *additional_input*).

   Note that the DRBG within the RBG2(P) construction **must** be reseeded before generating output. This is accomplished by requesting prediction resistance (i.e., by setting *prediction_resistance_request* = TRUE). See Requirement 17 in Section 4.4.

   b) If the randomness source is an RBG3(XOR) construction (see Figure 14), the **Get_randomness-source_input** call is replaced by:

   (*status*, *randomness-source_input*) = **RBG3(XOR)_Generate**(*RBG3_state_handle*, 3*s*/2, *prediction_resistance_request*, *additional_input*).

   A request for prediction resistance from the DRBG used by the RBG3(XOR) construction is optional.

   c) If the randomness source is an RBG3(RS) construction (see Figure 14), the **Get_randomness_-sourceinput** call is replaced by:

   (*status*, *randomness-source_input*) = **RBG3(RS)_Generate**(*RBG3_state_handle*, 3*s*/2, *additional_input*).

4.2.2. Requesting Pseudorandom Bits

Pseudorandom bits from the RBG1 construction **shall** be requested using the following call:

(*status*, *returned_bits*) = **Generate_function**(*RBG1_state_handle*, *requested_number_of_bits, s, prediction_resistance_request* = FALSE, *additional_input*).

The *prediction_resistance_request* is set to FALSE or the parameter may be omitted since a reseeding capability is not included in an RBG1 construction.

4.3. Using an RBG1 Construction with Subordinate DRBGs (Sub-DRBGs)

Figure 15 depicts an example of the use of optional subordinate DRBGs (sub-DRBGs) within the security boundary of an RBG1 construction. The RBG1 construction is used as the randomness source to provide separate outputs to instantiate each of its sub_DRBGs.

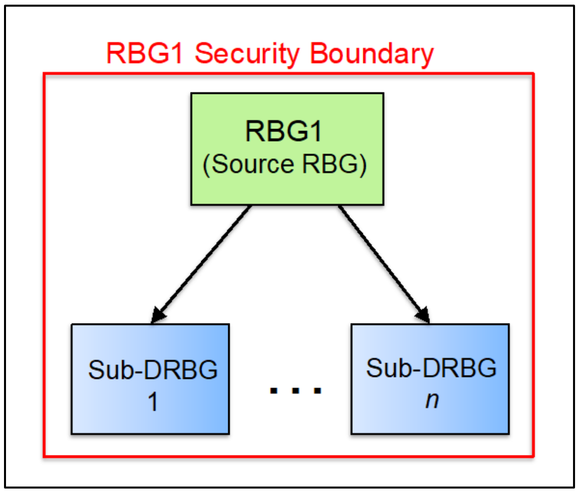

Fig. 15. RBG1 Construction with Sub-DRBGs

The RBG1 construction and each of its sub-DRBGs **shall** be implemented as separate physical or logical entities (see Figure 15).

- When implemented as separate physical entities, the DRBG algorithms used by the RBG1 construction and a sub-DRBG **shall** be the same DRBG algorithm (e.g., the RBG1 construction and all of its sub_DRBGs use HMAC_DRBG and SHA-256).
- When implemented as separate logical entities, the same software or hardware implementation of a DRBG algorithm is used but with a different internal state for each logical entity (e.g., the RBG1 construction has an internal state whose state handle is *RBG1_state_handle*, while the state handle for Sub-DRBG 1's internal state is *sub-DRBG1_state_handle*).

The sub-DRBGs have the following characteristics:

1. A sub-DRBG cannot be reseeded or provide prediction resistance.
2. Sub-DRBG outputs are considered outputs from the RBG1 construction.
3. The security strength that can be provided by a sub-DRBG is no more than the security strength of its randomness source (i.e., the RBG1 construction).
4. Each sub-DRBG has restrictions on its use (e.g., the number of outputs) as specified for its DRBG algorithm in [SP800-90A].
5. Sub-DRBGs cannot provide output with full entropy.
6. The number of sub-DRBGs that can be instantiated by a RBG1 construction is limited only by practical considerations associated with the implementation or application.

4.3.1. Instantiating a Sub-DRBG

Instantiation of the sub-DRBG is requested (e.g., by a consuming application) using the **Instantiate_function** discussed in Section 2.8.1.1 and [SP800-90A].

(*status*, *sub-DRBG_state_handle*) =
Instantiate_function(*s, prediction_resistance_flag* = FALSE*, personalization_string*),

where *s* is the requested security strength for the (target) sub-DRBG (note that *s* **must** be no greater than the security strength of the RBG1 construction).[15]

The (target) sub-DRBG is instantiated as follows:

1. When the sub-DRBG uses CTR_DRBG without a derivation function, *s* + 128 bits[16] **shall** be obtained from the RBG1 construction as follows:

   (*status*, *randomness-source_input*) = **Generate_function**(*RBG1_state_handle*, *s* + 128, *s*, *prediction_resistance_request* = FALSE, *additional_input*).

2. When the sub-DRBG uses any other DRBG (including a CTR_DRBG with a derivation function), 3*s*/2 bits **shall** be obtained from the RBG1 construction as follows:

   (*status*, *randomness-source_input*) = **Generate_function**(*RBG1_state_handle*, 3*s*/2, *s*, *prediction_resistance_request* = FALSE, *additional_input*).

4.3.2. Requesting Random Bits

Pseudorandom bits may be requested from a sub-DRBG using the following call (see Section 2.8.1.2):

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

where *sub_DRBG_state_handle* (if used) was returned by the **Instantiate_function** (see Sections 2.8.1.1 and 4.3.1).

4.4. Requirements

4.4.1. RBG1 Requirements

An RBG1 construction being instantiated has the following testable requirements (i.e., testable by the validation labs):

1. An **approved** DRBG from [SP800-90A] whose components are capable of providing the targeted security strength for the RBG1 construction **shall** be employed.

[15] The implementation is required to check the requested security strength (for the sub-DRBG) against the security strength recorded in the internal state of the RBG1's DRBG (see SP 800-90A).

[16] For AES, the block length is 128 bits, and the key length is equal to the security strength *s*. SP 800-90A requires the randomness input from the randomness source to be (key length + block length) bits when a derivation function is not used.

2. The RBG1 components **shall** be successfully validated for compliance with [SP800-90A], SP 800-90C, [FIPS140], and the specification of any other **approved** algorithm used within the RBG1 construction, as applicable.
3. The RBG1 construction **shall not** produce any output until it is instantiated.
4. The RBG1 construction **shall not** include a reseed capability.
5. The RBG1 construction **shall not** permit itself to be instantiated more than once.[17]
6. For a Hash_DRBG, HMAC_DRBG or CTR_DRBG (with a derivation function), $3s/2$ bits **shall** be obtained from a randomness source (see Requirements 13 - 17), where s is the targeted security strength for the DRBG used in the RBG1 construction.
7. For a CTR_DRBG (without a derivation function), $s + 128$ bits[18] **shall** be obtained from the randomness source (see Requirements 13 - 17), where s is the targeted security strength for the DRBG used in the RBG1 construction.
8. The internal state of the RBG1 construction **shall** be maintained[19] and updated to produce output on demand.
9. The RBG1 construction **shall not** provide output for generating requests that specify a security strength greater than the instantiated security strength of its DRBG.
10. If the RBG1 construction is used to instantiate a sub-DRBG, the RBG1 construction **may** directly produce output in addition to instantiating the sub-DRBG.
11. If the seedlife of the DRBG within the RBG1 construction is ever exceeded or a health test of the DRBG fails, the use of the RBG1 construction **shall** be terminated.
12. If a health test on the RBG1 construction fails, the RBG1 construction and all of its sub-DRBGs **shall** be terminated.

The non-testable requirements for the RBG1 construction are listed below. If these requirements are not met, no assurance can be obtained about the security of the implementation.

13. An **approved** RBG2(P) construction with support for prediction resistance requests or an RBG3 construction **must** be used as the randomness source for the DRBG in the RBG1 construction.
14. The randomness source **must** fulfill the requirements in Section 5 (for an RBG(P) construction) or Section 6 (for an RBG3 construction), as appropriate.
15. The randomness source **must** provide the requested number of bits at a security strength of s bits or higher, where s is the targeted security strength for the RBG1 construction.
16. The specific output of the randomness source (or portion thereof) that is used for the instantiation of an RBG1 construction **must not** be used for any other purpose, including for seeding a different instantiation.

[17] While technically possible to reseed the DRBG, doing so outside of very controlled conditions (e.g., "in the field") might result in seeds with less than the required amount of randomness.
[18] Note that $s + 128 = keylen + blocklen = seedlen$, as specified in SP 800-90A.
[19] This means ever-changing but maintained regardless of access to power for its entire lifetime.

17. If an RBG2(P) construction is used as the randomness source for the RBG1 construction, the RBG2(P) construction **must** be reseeded (i.e., prediction resistance must be obtained within the RBG2(P) construction) before generating bits for each RBG1 instantiation.
18. A physically secure channel **must** be used to insert the randomness input from the randomness source into the DRBG of the RBG1 construction.
19. An RBG1 construction **must not** be used for applications that require a higher security strength than has been instantiated.

4.4.2. Sub-DRBG Requirements

A sub-DRBG has the following testable requirements (i.e., testable by the validation labs).

1. The randomness source for a sub-DRBG **shall** be an RBG1 construction; a sub-DRBG **shall not** serve as a randomness source for another sub-DRBG.
2. A sub-DRBG **shall** employ the same DRBG components as its randomness source.
3. A sub-DRBG **shall** reside in the same security boundary as the RBG1 construction that initializes it.
4. The RBG1 construction **shall** fulfill the appropriate requirements of Section 4.4.1.
5. A sub-DRBG **shall** exist only for a limited time and purpose, as determined by the application or developer.
6. The output from the RBG1 construction that is used for sub-DRBG instantiation **shall not** be output from the security boundary of the construction and **shall not** be used for any other purpose, including for seeding a different sub-DRBG.
7. A sub-DRBG **shall not** permit itself to be instantiated more than once.
8. A sub-DRBG **shall not** provide output for use by the RBG1 construction (e.g., as additional input) or another sub-DRBG in the security boundary.
9. The security strength s requested for a target sub-DRBG instantiation **shall not** exceed the security strength that is supported by the RBG1 construction.
10. For a Hash_DRBG, HMAC_DRBG or CTR_DRBG (with a derivation function), $3s/2$ bits **shall** be obtained from the RBG1 construction for instantiation, where s is the requested security strength for the target sub-DRBG.

11 For a CTR_DRBG (without a derivation function), $s + 128$ bits **shall** be obtained from the RBG1 construction for instantiation, where s is the requested security strength for the target sub-DRBG.

12. A sub-DRBG **shall not** produce output until it is instantiated.
13. A sub-DRBG **shall not** provide output for generating requests that specify a security strength greater than the instantiated security strength of the sub-DRBG.
14. A sub-DRBG **shall** not include a reseed capability.

15. If the seedlife of a sub-DRBG is ever exceeded or a health test of the sub-DRBG fails, the use of the sub-DRBG **shall** be terminated.

A non-testable requirement for a sub-DRBG (not testable by the validation labs) is:

16. The output of a sub-DRBG **must not** be used as input to seed other DRBGs (e.g., the DRBGs in other RBGs).

5. RBG2 Constructions Based on Physical and/or Non-Physical Entropy Sources

An RBG2 construction is a cryptographically secure RBG with continuous access to one or more validated entropy sources within its RBG security boundary. The RBG is instantiated before use, generates outputs on demand, and can be used in an RBG3 construction (see Section 6). An RBG2 construction **may** support reseeding and may provide prediction resistance during generation requests (i.e., by performing a reseed of the DRBG prior to generating output). Both reseeding and providing prediction resistance are optional for this construction.

If full-entropy output is required by a consuming application, an RBG3 construction from Section 6 needs to be used rather than an RBG2 construction.

An RBG2 construction may be useful for all devices in which an entropy source can be implemented.

5.1. RBG2 Description

The DRBG for an RBG2 construction is contained within the same RBG security boundary and cryptographic module as its validated entropy source(s) (see Figure 16). The entropy source is used to provide the entropy bits for both DRBG instantiation and the reseeding of the DRBG used by the construction (e.g., to provide prediction resistance). An optional recommended personalization string and optional additional input may be provided from within the cryptographic module or from outside of that module.

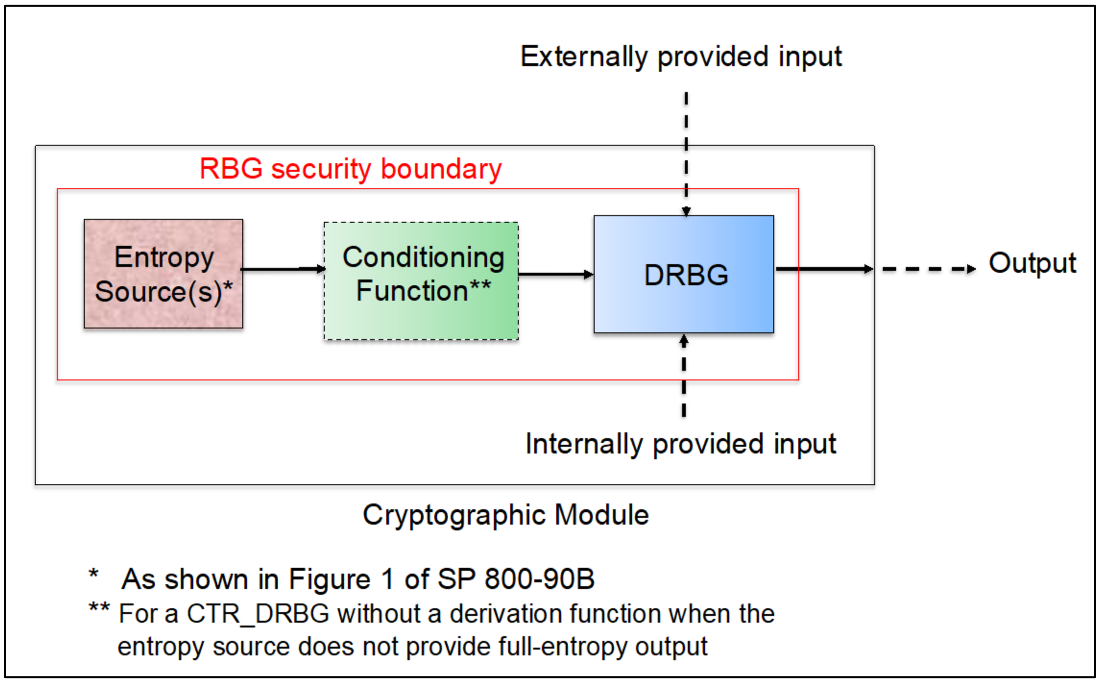
Externally provided input
RBG security boundary
Entropy Source(s)*
Conditioning Function**
DRBG
Output
Internally provided input
Cryptographic Module
* As shown in Figure 1 of SP 800-90B
** For a CTR_DRBG without a derivation function when the entropy source does not provide full-entropy output

Fig. 16. RBG2 Construction

The output from the RBG may be used within the cryptographic module or by an application outside of the module.

An example of an RBG2 construction is provided in Appendix B.4.

An RBG2 construction may be implemented to use one or more validated physical and/or non-physical entropy sources for instantiation and reseeding. Two variants of the RBG2 construction may be implemented.

1. An RBG2(P) construction uses the output of one or more validated physical entropy sources and (optionally) one or more validated non-physical entropy sources as discussed in Method 1 of Section 2.3 (i.e., only the entropy produced by validated physical entropy sources is counted toward the entropy required for instantiating or reseeding the RBG). Any amount of entropy may be obtained from a non-physical entropy source as long as sufficient entropy has been obtained from the physical entropy sources to fulfill an entropy request.
2. An RBG2(NP) construction uses the output of any validated non-physical or physical entropy sources as discussed in Method 2 of Section 2.3 (i.e., the entropy produced by both validated physical and non-physical entropy sources is counted toward the entropy required for instantiating or reseeding the RBG).

These variants affect the implementation of a **Get_ES_Bitstring** function (as specified in Section 2.8.2.2 and discussed in Section 3.1), either accessing the entropy source directly or via the **Get_conditioned_full_entropy_input** function during instantiation and reseeding (see Sections 5.2.1 and 5.2.3). That is, when instantiating and reseeding an RBG2(P) construction (including a DRBG within an RBG3 construction as discussed in Section 6), Method 1 in Section 2.3 is used to combine the entropy from the entropy sources, and Method 2 is used when instantiating and reseeding an RBG2(NP) construction.

5.2. Conceptual Interfaces

The RBG2 construction interfaces to the DRBG include function calls for instantiating the DRBG (see Section 5.2.1), generating pseudorandom bits on request (see Section 5.2.2), and (optionally) reseeding the DRBG at the end of the DRBG's seedlife and providing prediction resistance upon request (see Section 5.2.3).

Once instantiated, an RBG2 construction with a reseed capability may be reseeded on demand or whenever sufficient entropy is available.

5.2.1. RBG2 Instantiation

An RBG2 construction may be instantiated at any valid[20] security strength possible for the DRBG and its components using the following call:

(*status*, *RBG2_state_handle*) = **Instantiate_function** (*s, prediction_resistance_flag, personalization_string*),

[20] A security strength of either 128, 192, or 256 bits.

where s is the requested instantiation security strength for the DRBG. The *prediction_resistance_flag* (if used) is set to TRUE if prediction resistance is to be supported and FALSE otherwise.

An RBG2 construction obtains entropy for its DRBG from one or more validated entropy sources, either directly or using a conditioning function to process the output of the entropy source to obtain a full-entropy bitstring for instantiation (e.g., when employing a CTR_DRBG without a derivation function using entropy sources that do not provide full-entropy output).

SP 800-90A uses a **Get_randomness-source_input** call to obtain the entropy needed for instantiation (see SP 800-90A).

1. When the DRBG is a CTR_DRBG without a derivation function, full-entropy bits **shall** be obtained as follows:

   a) If the entropy source provides full-entropy output, the **Get_randomness-source_input** call is replaced by:[21, 22]

      (*status*, *entropy_bitstring*) = **Get_ES_Bitstring** (s + 128).[23]

      For an RBG2(P) construction, only validated physical entropy sources **shall** be used. The output of the entropy sources **shall** be concatenated to obtain the s + 128 full-entropy bits to be returned as *entropy_bitstring*.

      (This recommendation assumes that non-physical entropy sources cannot provide full-entropy output. Therefore, the **Get_ES_bitstring** function **shall not** be used with non-physical entropy sources in this case.)

   b) If the entropy sources does not provide full-entropy output, the **Get_randomness-source_input** call is replaced by:[24, 25]

      (*status*, *Full_entropy_bitstring*) =
      Get_conditioned_full_entropy_input(s + 128).

      Validated physical and/or non-physical entropy sources **shall** be used to provide the requested entropy. For an RBG2(P) construction, the requested s + 128 bits of entropy **shall** be counted as specified in Method 1 of Section 2.3. For an RBG2(NP) construction, the requested s + 128 bits of entropy **shall** be counted as specified in Method 2 of Section 2.3.

2. For the Hash_DRBG, HMAC_DRBG and CTR_DRBG (with a derivation function), the entropy source **shall** provide $3s/2$ bits of entropy to establish the security strength.

   a) If the consuming application requires full entropy in the returned bitstring, the **Get_randomness-source_input** call is replaced by:

      (*status*, *Full_entropy_bitstring*) =

      Get_conditioned_full_entropy_input($3s/2$).

[21] Appropriate changes may be required for the **Instantiate_function** in [SP800-90A] and the algorithms in Section 10 of that document.
[22] See Section 3.8.2.2 for a specification of the **Get_ES_Bitstring** function.
[23] For a CTR_DRBG using AES, s + 128 = the length of the key + the length of the AES block = *seedlen* (see Table 2 in SP 800-90A).
[24] Appropriate changes may be required for the **Instantiate_function** in [SP800-90A] and the algorithms in Section 10.2 of that document.
[25] See Section 4.3.2 for a specification of the **Get_conditioned_full_entropy_input** function.

b) If the consuming application does not require full entropy in the returned bitstring, the **Get_randomness-source_input** call is replaced by:

(*status*, *entropy_bitstring*) = **Get_ES_Bitstring**(3*s*/2).

Validated physical and/or non-physical entropy sources **shall** be used to provide the requested entropy. For an RBG2(P) construction, the requested 3*s*/2 bits of entropy **shall** be counted as specified in Method 1 of Section 2.3. For an RBG2(NP) construction, the requested 3*s*/2 bits of entropy **shall** be counted as specified in Method 2 of Section 3.3.

5.2.2. Requesting Pseudorandom Bits from an RBG2 Construction

Pseudorandom bits may be requested using the following call (see Section 2.8.1.2):

(*status*, *returned_bits*) = **Generate_function**(*RBG2_state_handle*, *requested_number_of_bits, requested_security_strength, prediction_resistance_request, additional_input*),

where *state_handle* (if used) was returned by the **Instantiate_function** (see Sections 2.8.1.1 and 5.2.1).

Support for prediction resistance is optional. If prediction resistance is supported, its use is optional. This RBG may be designed to always provide prediction resistance, to only provide prediction resistance upon request, or to be unable to provide prediction resistance (i.e., to not support prediction-resistance requests during generation).

Note that when prediction resistance is requested, the **Generate_function** will invoke the **Reseed_function**. If sufficient entropy is not available for reseeding, an error indication **shall** be returned, and the requested bits **shall not** be generated.

5.2.3. Reseeding an RBG2 Construction

As discussed in Section 2.4.2, when the RBG2 construction includes a reseed capability, the reseeding of the DRBG may be performed 1) upon request from a consuming application (either an explicit request for reseeding or a request for the generation of bits with prediction resistance); 2) on a fixed schedule based on time, number of outputs, or events; or 3) as sufficient entropy becomes available.

An RBG2 construction is reseeded using the following call:

status = **Reseed_function**(*RBG2_state_handle*, *additional_input*),

where the *RBG2_state_handle* (when used) was obtained during the instantiation of the RBG (see Sections 2.8.1.1 and 5.2.1).

SP 800-90A uses a **Get_randomness-source_input** call to obtain the entropy needed for reseeding the DRBG (see Section 2.8.1.3 herein and in [SP800-90A]. The DRBG is reseeded at the instantiated security strength recorded in the DRBG's internal state. The **Get_randomness-source_input** call in SP 800-90A **shall** be replaced with the following:

1. For the CTR_DRBG without a derivation function, use the appropriate replacement as specified in step 1 of Section 5.2.1.

2. For the Hash_DRBG, HMAC_DRBG and CTR_DRBG (with a derivation function), replace the **Get_randomness-sourceinput** call in the **Reseed_function** with the following:[26]

   a) If the consuming application requires full entropy in the returned bitstring, the **Get_randomness-source_input** call is replaced by:

      (*status*, *Full_entropy_bitstring*) = **Get_conditioned_full_entropy_input**(*s*).

   b) If the consuming application does not require full entropy in the returned bitstring, the **Get_randomness-source_input** call is replaced by:

      (*status*, *entropy_bitstring*) = **Get_ES_Bitstring(***s*).

   Validated physical and/or non-physical entropy sources **shall** be used to provide the requested entropy. For an RBG2(P) construction, the requested *s* bits of entropy **shall** be counted as specified in Method 1[27] of Section 2.3. For an RBG2(NP) construction, the requested *s* bits of entropy **shall** be counted as specified in Method 2[28] of Section 2.3.

5.3. RBG2 Requirements

An RBG2 construction has the following requirements in addition to those specified in [SP800-90A]:

1. The RBG **shall** employ an **approved** and validated DRBG from [SP800-90A] whose components are capable of providing the targeted security strength for the RBG.
2. The RBG and its components **shall** be successfully validated for compliance with [SP800-90A], [SP800-90B], SP 800-90C, [FIPS140], and the specification of any other **approved** algorithm used within the RBG, as appropriate.
3. The RBG **may** include a reseed capability. If implemented, the reseeding of the DRBG **shall** be performed either a) upon request from a consuming application (either an explicit request for reseeding or a request for the generation of bits with prediction resistance); b) on a fixed schedule based on time, number of outputs, or events; and/or c) as sufficient entropy becomes available.
4. Validated entropy sources **shall** be used to instantiate and reseed the DRBG. A non-validated entropy sources **shall not** be used for this purpose.
5. The entropy sources used for the instantiation and reseeding of an RBG(P) construction **shall** include one or more validated physical entropy sources; the inclusion of one or more validated non-physical entropy sources is optional. A bitstring that contains entropy **shall** be assembled and the entropy in that bitstring determined as specified in Method 1 of Section 2.3 (i.e., only the entropy provided by validated physical entropy sources **shall** be counted toward fulfilling the amount of entropy in an entropy request).

[26] See Sections 2.8.2.2 and 3.1 for discussions of the Get_ES_bitstring function.

[27] Method 1 only counts the entropy provided by validated physical sources.

[28] Method 2 counts the entropy provided by both physical and non-physical entropy sources.

6. The entropy sources used for the instantiation and reseeding of an RBG2(NP) construction **shall** include one or more validated non-physical entropy sources; the inclusion of one or more validated physical entropy sources is optional. A bitstring containing entropy **shall** be assembled and the entropy in that bitstring determined as specified in Method 2 of Section 2.3 (i.e., the entropy provided by both validated non-physical entropy sources and any validated physical entropy sources included in the implementation **shall** be counted toward fulfilling the requested amount of entropy).
7. The DRBG **shall** be capable of being instantiated and reseeded at the maximum security strength (*s*) for the DRBG design (see [SP800-90A]).
8. A specific entropy-source output (or portion thereof) **shall not** be reused (e.g., it is destroyed after use).
9. When instantiating and reseeding a CTR_DRBG without a derivation function, (s + 128) bits with full entropy **shall** be obtained either directly from the entropy source or from the entropy source via an external vetted conditioning function (see Section 3.3).
10. For a Hash_DRBG, HMAC_DRBG or CTR_DRBG (with a derivation function), a bitstring with at least $3s/2$ bits of entropy **shall** be obtained from the entropy source to instantiate the DRBG at a security strength of *s* bits. When reseeding is performed, a bitstring with at least *s* bits of entropy **shall** be obtained from the entropy source.
11. The DRBG **shall** be instantiated before first use (i.e., before providing output for use by a consuming application) and reseeded using the validated entropy sources used for instantiation.
12. When health tests detect the failure of a validated entropy source, the failure **shall** be handled as discussed in Section 7.1.2.1.

A non-testable requirement for the RBG (not testable by the validation labs) is:

13. The RBG **must not** be used by applications that require a higher security strength than has been instantiated in the DRBG.

6. RBG3 Constructions Based on Physical Entropy Sources

An RBG3 construction is designed to provide full entropy (i.e., an RBG3 construction can support all security strengths). The RBG3 constructions specified in this Recommendation include one or more entropy sources and an **approved** DRBG from SP 800-90A that can and will be instantiated at a security strength of 256 bits. If an entropy source fails in an undetected manner, the RBG continues to operate as an RBG2(P) construction, providing outputs at the security strength of its DRBG (256 bits) (see Section 5 and Appendix A). If a failure is detected, the RBG operation **shall** be terminated.

Two RBG3 constructions are specified:

1. RBG3(XOR) – This construction is based on combining the output of one or more validated entropy sources with the output of an instantiated, **approved** DRBG using an exclusive-or operation (see Section 6.2).
2. RBG3(RS) – This construction is based on using one or more validated entropy sources to continuously reseed the DRBG (see Section 6.3).

An RBG3 construction continually accesses its entropy sources, and its DRBG may be reseeded whenever requested (e.g., to provide prediction resistance for the DRBG's output). Upon receipt of a request for random bits from a consuming application, the entropy source is accessed to obtain sufficient bits for the request. See Sections 3.1 and 3.2 for further discussion about accessing the entropy source(s).

An implementation may be designed so that the DRBG implementation used within an RBG3 construction can be directly accessed by a consuming application (i.e., the directly accessible DRBG uses the same internal state as the RBG3 construction).

An RBG3 construction is useful when bits with full entropy are required or a higher security strength than RBG1 and RBG2 constructions can support is needed.

6.1. General Requirements

RBG3 constructions have the following general security requirements. See Sections 6.2.2 and 6.3.2 for additional requirements for the RBG3(XOR) and RBG3(RS) constructions, respectively.

1. An RBG3 construction **shall** be designed to provide outputs with full entropy using one or more validated independent physical entropy sources as specified for Method 1 in Section 3.3 (i.e., only the entropy provided by validated physical entropy sources **shall** be counted toward fulfilling entropy requests, although entropy provided by any validated non-physical entropy source may be used but not counted).
2. An RBG3 construction and its components **shall** be successfully validated for compliance with the corresponding requirements in [SP800-90A], [SP800-90B], SP 800-90C, [FIPS 140] and the specification of any other **approved** algorithm used within the RBG, as appropriate.
3. The DRBG within the RBG3 construction **shall** be capable of supporting a security strength of 256 bits (i.e., a CTR_DRBG based on AES-256 or either Hash_DRBG or HMAC_DRBG using a hash function with an output length of at least 256 bits).

4. The DRBG **shall** be instantiated at a security strength of 256 bits before the first use of the RBG3 construction or direct access of the DRBG.
5. The DRBG **shall** include a reseed function to support reseed requests.
6. A specific entropy-source output (or portion thereof) **shall not** be reused (e.g., the same entropy-source outputs **shall not** be used for an RBG3 request and a request to a separate instantiation of a DRBG).
7. If the DRBG is directly accessible, the requirements in Section 5.3 for RBG2(P) constructions **shall** apply to the direct access of the DRBG.
8. When health tests detect the failure of a validated physical entropy source, the failure **shall** be handled as discussed in Section 7.1.2.1. If a failure is detected in a non-physical entropy source, the consuming application **shall** be notified.

6.2. RBG3(XOR) Construction

An RBG3(XOR) construction contains one or more validated entropy sources and a DRBG whose outputs are XORed to produce full-entropy output (see Figure 17). In order to provide the required full-entropy output, the input to the XOR (shown as "⊕" in the figure) from the entropy-source side of the figure **shall** consist of bits with full entropy (see Section 2.1).[29] If the entropy sources cannot provide full-entropy output, then an external conditioning function **shall** be used to condition the output of the entropy sources to a full-entropy bitstring before XORing with the output of the DRBG (see Section 3.3).

[29] Note that the DRBGs themselves are not designed to inherently provide full-entropy output.

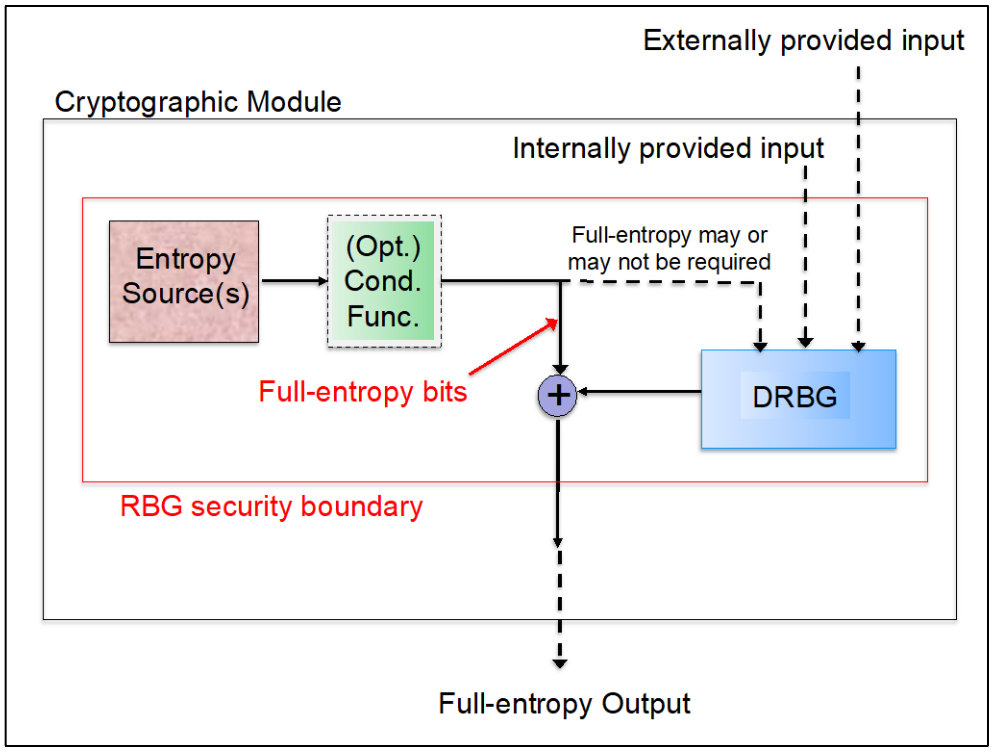

Fig. 17. RBG3(XOR) Construction

When n bits of output are requested from an RBG3(XOR) construction, n bits of output from the DRBG are XORed with n full-entropy bits obtained either directly from the entropy source or from the entropy source after cryptographic processing by an external vetted conditioning function (see Section 3.3). When the entropy source is working properly,[30] an n-bit output from the RBG3(XOR) construction is said to provide n bits of entropy or to support a security strength of n bits. The DRBG used in the RBG3(XOR) construction is always required to support a 256-bit security strength. If the entropy source fails without being detected and the DRBG has been successfully instantiated with at least 256 bits of entropy, the DRBG continues to produce output at a security strength of 256 bits.

An example of an RBG3(XOR) design is provided in Appendix B.5.

6.2.1. Conceptual Interfaces

The RBG interfaces include function calls for instantiating the DRBG (see Section 6.2.1.1), generating random bits on request (see Section 6.2.1.2), and reseeding the DRBG instantiation(s) (see Section 6.2.1.3).

6.2.1.1. Instantiation of the DRBG

The DRBG for the RBG3(XOR) construction is instantiated as follows:

[30] The entropy source provides at least the amount of entropy determined during the entropy-source validation process.

RBG3(XOR)_DRBG_Instantiate:

```
Input: integer (prediction_resistance_flag), string personalization_string.

Output: integer status, integer state_handle.

Process:
    1. (status, RBG3(XOR)_state_handle) = Instantiate_function(256,
       prediction_resistance_flag, personalization_string).

    2. Return (status, RBG3(XOR)_state_handle).
```

In step 1, the DRBG is instantiated at a security strength of 256 bits. The *prediction_resistance_flag* and *personalization_string* (when provided as input to the **RBG3(XOR)_DRBG_Instantiate** function) **shall** be used in step 1.

In step 2, the *status* and *RBG3(XOR)_state_handle* that were obtained in step 1 are returned. Note that if the *status* does not indicate a successful instantiate process (i.e., a failure is indicated), the returned state handle **shall** be invalid (e.g., a *Null* value). The handling of status codes is discussed in Section 2.8.3.

6.2.1.2. Random and Pseudorandom Bit Generation

Let *n* be the requested number of bits to be generated, and let the *RBG3(XOR)_state_handle* be the value returned by the instantiation function for RBG3's DRBG instantiation (see Section 6.2.1.1). Random bits with full entropy **shall** be generated by the RBG3(XOR) construction using the following generate function:

RBG3(XOR)_Generate:

```
Input: integer (RBG3(XOR)_state_handle, n, prediction_resistance_request), string
    additional_input.

Output: integer status, string returned_bits.

Process:
    1. (status, ES_bits) = Request_entropy(n).

    2. If (status ≠ SUCCESS), then return (status, invalid_string).

    3. (status, DRBG_bits) = Generate_function(RBG3(XOR)_state_handle, n, 256,
       prediction_resistance_request, additional_input).

    4. If (status ≠ SUCCESS), then return (status, invalid_string).

    5. returned_bits = ES_bits ⊕ DRBG_bits.

    6. Return (SUCCESS, returned_bits).
```

Step 1 requests that the entropy sources generate bits. Since full-entropy bits are required, the (place holder) **Request_entropy** call **shall** be replaced by one of the following:

- If full-entropy output is provided by all validated physical entropy sources used by the RBG3(XOR) implementation, and non-physical entropy sources are not used,[31] step 1 becomes:

  (*status*, *ES_bits*) = **Get_ES_Bitstring**(*n*).

  The **Get_ES_Bitstring** function[32] **shall** use Method 1 in Section 2.3 to obtain the *n* full-entropy bits that were requested in order to produce the *ES_bits* bitstring.

- If full-entropy output is not provided by all physical entropy sources, or the output of both physical and non-physical entropy sources is also used by the implementation, step 1 becomes:

  (*status*, *ES_bits*) = **Get_conditioned_full_entopy_input**(*n*).

  The **Get_conditioned_full_entropy_input** construction is specified in Section 3.3.2. It requests entropy from the entropy sources in step 3.1 of that construction with a **Get_ES_Bitstring** call. The **Get_ES_Bitstring** call **shall** use Method 1 (as specified in Section 3.3) when collecting the output of the entropy sources (i.e., only the entropy provided by physical entropy sources is counted).

In step 2, if the request in step 1 is not successful, abort the **RBG3(XOR)_Generate** function, returning the *status* received in step 1 and an invalid bitstring as the *returned_bits* (e.g., a *Null* bitstring). If *status* indicates a success, *ES_bits* is the full-entropy bitstring to be used in step 5.

In step 3, the RBG3(XOR)'s DRBG instantiation is requested to generate *n* bits at a security strength of 256 bits. The DRBG instantiation is indicated by the *RBG3(XOR)_state_handle*, which was obtained during instantiation (see Section 6.2.1.1). If a prediction-resistance request and/or additional input are provided in the **RBG.3(XOR)_Generate** call, they **shall** be included in the **Generate_function** call.

Note that it is possible that the DRBG would require reseeding during the **Generate_function** call in step 3 (e.g., because of a prediction-resistance request, or the end of the seedlife of the DRBG has been reached). If a reseed of the DRBG is required during **Generate-function** execution, the DRBG **shall** be reseeded as specified in Section 6.2.1.3 with bits not otherwise used by the RBG.

In step 4, if the **Generate_function** request is not successful, the **RBG3(XOR)_Generate** function is aborted, and the *status* received in step 3 and an invalid bitstring (e.g., a *Null* bitstring) are returned to the consuming application. If *status* indicates a success, *DRBG_bits* is the pseudorandom bitstring to be used in step 5.

Step 5 combines the bitstrings returned from the entropy sources (from step 1) and the DRBG (from step 3) using an XOR operation. The resulting bitstring is returned to the consuming application in step 6.

[31] Since non-physical entropy sources are assumed to be incapable of providing full-entropy output, they cannot contribute to the bitstring provided by the **Get_ES_Bitstring** function.
[32] See Section 3.10.2.2.

6.2.1.3. Pseudorandom Bit Generation Using a Directly Accessible DRBG

Pseudorandom bit generation by a direct access of the DRBG is accomplished as specified in Section 5.2.2 using the state handle obtained during instantiation (see Section 6.2.1.1).

When directly accessing the DRBG instantiation that is also used by the RBG3(XOR) construction, the following function is used:

(*status*, *returned_bits*) = **Generate_function**(*RBG3(XOR)_state_handle*, *requested_number_of_bits, requested_security_strength, prediction_resistance_request, additional_input*),

where:

- *RBG3(XOR)_state_handle* indicates the DRBG instantiation to be used.
- *requested_security_strength* ≤ 256.
- *prediction-resistance-request* is either TRUE or FALSE; requesting prediction resistance during the **Generate_function** is optional.
- The use of additional input is optional.

Note that when prediction resistance is requested, the **Generate_function** will invoke the **Reseed_function** (see Section 6.2.1.3). If sufficient entropy is not available for reseeding, an error indication **shall** be returned, and the requested bits **shall not** be generated.

6.2.1.4. Reseeding the DRBG Instantiations

Reseeding is performed using the entropy sources in the same manner as an RBG2 construction using the appropriate state handle (e.g., *RBG3(XOR)_state_handle,* as specified in Section 6.2.1.1).

6.2.2. RBG3(XOR) Requirements

An RBG3(XOR) construction has the following requirements in addition to those provided in Section 6.2:

1. Bitstrings with full entropy **shall** be provided to the XOR operation either directly from the concatenated output of one or more validated physical entropy sources or by an external conditioning function using the output of one or more validated entropy sources as specified in Method 1 of Section 2.3. In the latter case, the output of validated non-physical entropy sources may be used without counting any entropy that they might provide.
2. The same entropy-source outputs used by the DRBG for instantiation or reseeding **shall not** be used as input into the RBG's XOR operation.
3. The DRBG instantiations **shall** be reseeded occasionally (e.g., after a predetermined period of time or number of generation requests).

6.3. RBG3(RS) Construction

The second RBG3 construction specified in this document is the RBG3(RS) construction shown in Figure 18, and an example of this construction is provided in Appendix B.6.

Note that external conditioning of the outputs from the entropy sources during instantiation and reseeding is required when the DRBG is a CTR_DRBG without a derivation function and the entropy sources do not provide a bitstring with full entropy.

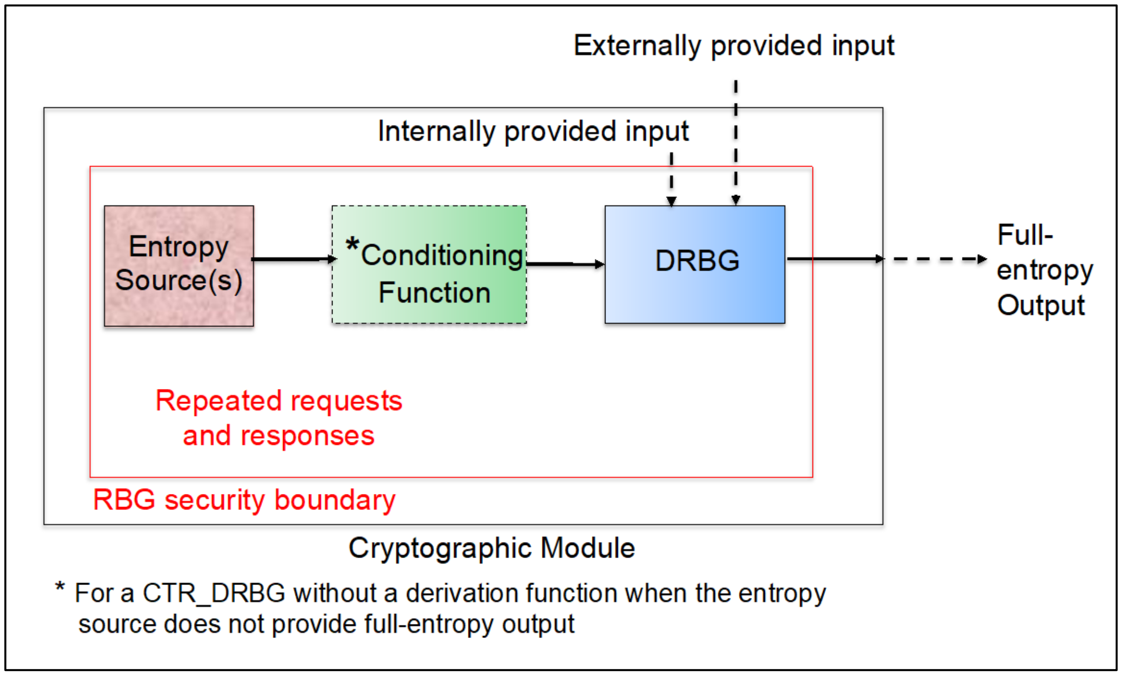
Externally provided input
Internally provided input
Entropy Source(s)
*Conditioning Function
DRBG
Full-entropy Output
Repeated requests and responses
RBG security boundary
Cryptographic Module
* For a CTR_DRBG without a derivation function when the entropy source does not provide full-entropy output

Fig. 18. RBG3(RS) Construction

6.3.1. Conceptual Interfaces

The RBG interfaces include function calls for instantiating the DRBG (see Section 6.3.1.1), generating random bits on request (see Section 6.3.1.2), and reseeding the DRBG instantiation (see Section 6.3.1.3).

6.3.1.1. Instantiation of the DRBG Within an RBG3(RS) Construction

DRBG instantiation is performed as follows:

RBG3(RS)_DRBG_Instantiate:

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

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

Process:

1. (*status*, *RBG3(RS)_state_handle*) = **Instantiate_function**(256, *prediction_resistance_flag* = TRUE, *personalization_string*).

2. Return (*status*, *RBG3(RS)_state_handle*).

In step 1, the DRBG is instantiated at a security strength of 256 bits. The *prediction_resistance_flag* is set to TRUE, and *personalization_string* (when provided as input to the **RBG3(RS)_DRBG_Instantiate** function) **shall** be used in step 1.

In step 2, the *status* and the *RBG3(RS)_state_handle* are returned. Note that if the *status* does not indicate a successful instantiate process (i.e., a failure is indicated), the returned state handle **shall** be invalid (e.g., a *Null* value). The handling of status codes is discussed in Section 2.8.3.

6.3.1.2. Random and Pseudorandom Bit Generation

6.3.1.2.1 Generation Using the RBG3(RS) Construction

When an RBG3(RS) construction receives a request for *n* random bits, the DRBG instantiation used by the construction needs to be reseeded with sufficient entropy so that bits with full entropy can be extracted from the DRBG's output block.

Table_3 provides information for generating full-entropy output from the DRBGs in SP 800-90A that use the cryptographic primitives listed in the table. Each primitive in the table can support a security strength of 256 bits – the highest security strength recognized by this Recommendation. To use the table, select the row that identifies the cryptographic primitive used by the implemented DRBG.

- Column 1 lists the DRBGs.
- Column 2 identifies the cryptographic primitives that can be used by the DRBG(s) in column 1 to support a security strength of 256 bits.
- Column 3 indicates the length of the output block (*output_len*) for the cryptographic primitives in column 2.
- Column 4 indicates the amount of fresh entropy that is obtained by a **Reseed_function** when the **Generate_function** is invoked with prediction resistance requested.

Table 3. Values for generating full-entropy bits by an RBG3(RS) Construction

DRBG	**DRBG Primitives**	**Output Block Length (*output_len*) in bits**	**Entropy obtained during a normal reseed operation**
CTR_DRBG (with no derivation function)	AES-256	128	384
CTR_DRBG (using a derivation function)	AES-256	128	256
Hash_DRBG or HMAC_DRBG	SHA-256 SHA3-256	256	256
	SHA-384 SHA3-384	384	256
	SHA-512 SHA3-512	512	256

The strategy used for obtaining full-entropy output from the RBG3(RS) construction requires obtaining sufficient fresh entropy and subsequently extracting full entropy bits from the output block in accordance with item 11 of Section 2.6.

For the **RBG3(RS)_Generate** function:

- Let *n* be the requested number of full-entropy bits to be generated by an RBG3(RS) construction.
- Let *RBG3(RS)_state_handle* be a state handle returned from the instantiate function (see Section 6.3.1.1).

Random bits with full entropy **shall** be generated as follows:

RBG3(RS)_ Generate:

```
Input: integer (RBG3(RS)_state_handle, n), string additional_input.

Output: integer status, bitstring returned_bits.

Process:
        1. full-entropy_bits =Null.
        2. sum = 0.
        3. While (sum < n),
            3.1  Obtain generated_bits from the entropy source.
            3.2  If (status ≠ SUCCESS), then return (status, invalid_bitstring).
            3.3  full-entropy_bits = full_entropy_bits || generated_bits.
            3.4  sum = sum + len(generated_bits).
        4. Return (SUCCESS, leftmost(full-entropy_bits, n)).
```

In steps 1 and 2, the bitstring intended to collect the generated bits for returning to the calling application (i.e., *full-entropy_bits*) is initialized to the *Null* bitstring, and the counter for the number of bits obtained for fulfilling the request is initialized to zero.

Step 3 is iterated until *n* bits have been generated.

In step 3.1, the DRBG is requested to obtain sufficient entropy so that a bitstring with full entropy can be extracted from the output block. The form of the request depends on the DRBG algorithm used in the RBG3(RS) construction and the method for obtaining a full-entropy bitstring (see Section 2.6, item 11). Note that extracting fewer full-entropy bits from the DRBG's output block is permitted.

For a CTR_DRBG (with or without a derivation function), a maximum of 128 bits with full entropy can be provided from the AES output block for each iteration of the DRBG as follows:

(*status*, *generated_bits*) = **Generate_function**(*RBG3(RS)_state_handle,* 128, 256*, prediction_resistance_request* = TRUE*, additional_input*).

The **Generate_function** generates 128 (full entropy) bits after reseeding the CTR_DRBG with either 256 or 384 bits of entropy (by setting *prediction_resistance_request* = TRUE).[33]

For a hash-based DRBG (i.e., Hash_DRBG and HMAC_DRBG), a maximum of 256 full-entropy bits can be produced from each iteration of the DRBG as follows:

3.1.1 (*status*, *additional_entropy*) = **Get_ES_Bitstring** (64).

3.1.2 If (*status* ≠ SUCCESS*)*, then return (*status*, *invalid_bitstring*).

3.1.3 (*status*, *generated_bits*) = **Generate_function**(*RBG3(RS)_state_handle,* 256*,* 256, *prediction_resistance_request* = TRUE*, additional_input* || *additional_entropy*).

At least 64 bits of entropy beyond the amount obtained during reseeding are required. As shown in Table 3, the reseeding process will acquire 256 bits of entropy. The (256 + 64 = 384) bits of entropy are inserted into the DRBG by 1) obtaining a bitstring with at least 64 bits of entropy directly from the entropy sources (step 3.1.1), 2) concatenating the additional entropy bits with any *additional_input* provided in the **RBG3(RS)_Generate** call, and 3) requesting the generation of 256 bits with prediction resistance and including the concatenated bitstring. This results in both the reseed of the DRBG with 256 bits of entropy and the insertion of the additional 64 bits of entropy) (step 3.1.3).

For a hash-based DRBG (i.e., Hash_DRBG and HMAC_DRBG), a maximum of 192 full-entropy bits can be produced from each iteration of the DRBG as follows:

(*status*, *generated_bits*) = **Generate_function**(*RBG3(RS)_state_handle,* 192*,* 256*, prediction_resistance_request* = TRUE*, additional_input*).

The DRBG is reseeded with 256 bits of entropy by requesting generation with prediction resistance and extracting only (256 – 64 = 192) bits from the DRBG's output block as full-entropy bits.

In step 3.2, if the **Generate_function** request invoked in step 3.1 is not successful, the **RBG3(RS)_Generate** function is aborted, and the *status* received in step 3.1 and an invalid bitstring (e.g., a *Null* bitstring) are returned to the consuming application.

Step 3.3 combines the full-entropy bitstrings obtained in step 3.1 with previously generated full-entropy bits using a concatenation operation.

Step 3.4 adds the number of full-entropy bits produced in step 3.1 to those generated in previous iterations of step 3.

If *sum* is less than the requested number of bits (*n*), repeat step 3 starting at step 3.1.

In step 4, the leftmost *n* bits are selected from the collected bitstring (i.e., *full-entropy_bits*) and returned to the consuming application.

6.3.1.2.2 Generation Using a Directly Accessible DRBG

[33] The use of the *prediction_resistance_request* will handle the differences between the two versions of the CTR_DRBG (i.e., with or without a derivation function).

Direct access of the DRBG is accomplished as specified in Section 5.2.2 using the state handle associated with the instantiation and internal state that was returned for the DRBG (see Section 6.3.1.1).

(*status*, *returned_bits*) = **Generate_function**(*RBG3(RS)_state_handle*, *requested_number_of_bits, requested_security_strength, prediction_resistance_request, additional_input*),

where *state_handle* (if used) was returned by the **Instantiate_function** (see Section 6.3.1.1).

When the previous generate request was made to the RBG3(RS) construction rather than directly to the DRBG, the *prediction_resistance_request* parameter **shall** be set to TRUE. Otherwise, requesting prediction resistance during the **Generate_function** is optional.

6.3.1.3. Reseeding

Reseeding is performed during a **Generate_function** request to a directly accessible DRBG (see Section 6.3.1.2.2) when prediction resistance is requested or the end of the DRBG's seedlife is reached. The **Generate_function i**nvokes the **Reseed_function s**pecified in [SP800-90A].

Reseeding may also be performed on demand as specified in Section 4.2.3 using the *RBG3(RS)_state_handle* if provided during instantiation.

6.3.2. Requirements for a RBG3(RS) Construction

An RBG3(RS) construction has the following requirements in addition to those provided in Section 6.1:

1. Fresh entropy **shall** be acquired either directly from all independent validated entropy sources (see Section 3.2) or (in the case of a CTR_DRBG used as the DRBG when the entropy sources do not provide full-entropy output) from an external conditioning function that processes the output of the validated entropy sources as specified in Section 3.3.2. Method 1 in Section 2.3 **shall** be used when collecting the required entropy (i.e., only the entropy provided by validated physical entropy sources **shall** be counted toward fulfilling the amount of entropy requested).
2. If the DRBG is directly accessible, a reseed of the DRBG instantiation **shall** be performed before generating output in response to a request for output from the directly accessible DRBG when the previous use of the DRBG was by the RBG3(RS) construction. This could require an additional internal state value to record the last use of the DRBG for generation (e.g., used by an **RBG3(RS)_Generate** function as specified in Section 6.3.1.2.1 or directly accessed by a (DRBG) **Generate_function** as discussed in Section 6.3.1.2.2).

7. Testing

Two types of testing are specified in this Recommendation: health testing and implementation-validation testing. Health testing **shall** be performed on all RBGs that claim compliance with this Recommendation (see Section 7.1). Section 7.2 provides requirements for implementation validation.

7.1. Health Testing

Health testing is the testing of an implementation prior to and during normal operations 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).

An RBG **shall** support the health tests specified in [SP800-90A] and [SP800-90B] as well as perform health tests on the components of SP 800-90C (see Section 7.1.1). [FIPS 140] specifies the testing to be performed within a cryptographic module.

7.1.1. Testing RBG Components

Whenever an RBG receives a request to start up or perform health testing, a request for health testing **shall** be issued to the RBG components (e.g., the DRBG and any entropy source).

7.1.2. Handling Failures

Failures may occur during the use of entropy sources and during the operation of other components of an RBG.

Note that [SP800-90A] and [SP800-90B] discuss the error handling for DRBGs and entropy sources, respectively.

7.1.2.1. Entropy-Source Failures

A failure of a validated entropy source may be reported to the **Get_ES_Bitstring** function (see item 3 of Section 3.1 and item 4 of Section 3.2) during entropy requests to the entropy sources or to the RBG when the entropy sources continue to function when entropy is not requested (see item 5 of Section 3.2).

7.1.2.2. Failures by Non-Entropy-Source Components

Failures by non-entropy-source components may be caused by either hardware or software failures. Some of these may be detected using the health testing within the RBG using known-answer tests. Failures could also be detected by the system in or on which the RBG resides.

When such failures are detected that affect the RBG, RBG operation **shall** be terminated. The RBG **must not** be resumed until the reasons for the failure have been determined and the failures have been repaired and successfully tested for proper operation.

7.2. Implementation Validation

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

- Validating the components from [SP800-90A] and [SP800-90B].
- Validating the use of the constructions in SP 800-90C via code inspection, known-answer tests, or both, as appropriate.
- 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 testers that an RBG that claims compliance with this Recommendation has been implemented correctly. This documentation **shall** include the following as a minimum:

- An identification of the constructions and components used by the RBG, including a diagram of the interaction between the constructions and components.
- If an external conditioning function is used, an indication of the type of conditioning function and the method for obtaining any keys that are required by that function.
- Appropriate documentation, as specified in [SP800-90A] and [SP800-90B]. The DRBG and the entropy sources **shall** be validated for compliance with SP 800-90A or SP 800-90B, respectively, and the validations successfully finalized before the completion of RBG implementation validation.
- For an RBG1 or RBG2 construction, the maximum security-strength that can be supported by the DRBG.
- A description of all validated and non-validated entropy sources used by the RBG, including identifying whether the entropy source is a physical or non-physical entropy source.
- Documentation justifying the independence of all validated entropy sources from all other validated and non-validated entropy sources.
- An identification of the features supported by the RBG (e.g., access to the underlying DRBG of an RBG3 construction).
- 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 description of the RBG components within the RBG security boundary (see Section 2.5).
- For an RBG1 construction, a statement indicating that the randomness source **must** be a validated RBG2(P) or RBG3 construction (e.g., this could be provided in user documentation and/or a security policy).
- If sub-DRBGs can be used in an RBG1 construction, the maximum number of sub-DRBGs and the security strengths to be supported by the sub-DRBGs.

- For an RBG2 construction (including a directly accessible DRBG within an RBG3 construction), a statement indicating whether prediction resistance is always provided when a request is made by a consuming application, only provided when requested, or never provided.
- For an RBG3 construction, a statement indicating whether the DRBG can be accessed directly.
- Documentation specifying the guidance to users about fulfilling the non-testable requirements for RBG1 constructions, RBG2 constructions, and sub-DRBGs, as appropriate (see Sections 5.4 and 6.3, respectively).

References

[FIPS140]	National Institute of Standards and Technology (2001) *Security Requirements for Cryptographic Modules*. (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 140-2, Change Notice 2 December 03, 2002. https://doi.org/10.6028/NIST.FIPS.140-2
	National Institute of Standards and Technology (2010) *Security Requirements for Cryptographic Modules*. (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 140-3. https://doi.org/10.6028/NIST.FIPS.140-3
[FIPS140IG]	National Institute of Standards and Technology, Canadian Centre for Cyber Security *Implementation Guidance for FIPS 140-2 and the Cryptographic Module Validation Program*, [Amended]. Available at https://csrc.nist.gov/csrc/media/projects/cryptographic-module-validation-program/documents/fips140-2/FIPS1402IG.pdf
[FIPS180]	National Institute of Standards and Technology (2015) *Secure Hash Standard (SHS)*. (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 180-4. https://doi.org/10.6028/NIST.FIPS.180-4
[FIPS197]	National Institute of Standards and Technology (2001) *Advanced Encryption Standard (AES)*. (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 197. https://doi.org/10.6028/NIST.FIPS.197
[FIPS198]	National Institute of Standards and Technology (2008) *The Keyed-Hash Message Authentication Code (HMAC)*. (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 198-1. https://doi.org/10.6028/NIST.FIPS.198-1.
[FIPS202]	National Institute of Standards and Technology (2015) *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 202. https://doi.org/10.6028/NIST.FIPS.202
[NISTIR8427]	Buller D, Kaufer A, Roginsky AL, Sonmez Turan M (2022). Discussion on the Full Entropy Assumption of SP 800-90 Series. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Internal Report (NISTIR) 8427 ipd. https://doi.org/10.6028/NIST.IR.8427.ipd
[SP800-38B]	Dworkin MJ (2005) *Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authentication*. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication

(SP) 800-38B, Includes updates as of October 6, 2016. https://doi.org/10.6028/NIST.SP.800-38B

[SP800-57Part1] Barker EB (2020) Recommendation for Key Management: Part 1 – General. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-57 Part 1, Rev. 5. https://doi.org/10.6028/NIST.SP.800-57pt1r5

[SP800-67] Barker EB, Mouha N (2017*) Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Ciph*er. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-67, Rev. 2. https://doi.org/10.6028/NIST.SP.800-67r2

[SP800-90A] Barker EB, Kelsey JM (2015*) Recommendation for Random Number Generation Using Deterministic Random Bit Generato*rs. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-90A, Rev. 1. https://doi.org/10.6028/NIST.SP.800-90Ar1

[SP800-90B] Sönmez Turan M, Barker EB, Kelsey JM, McKay KA, Baish ML, Boyle M (2018*) Recommendation for the Entropy Sources Used for Random Bit Generati*on. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-90B. https://doi.org/10.6028/NIST.SP.800-90B

[SP800-131A] Barker EB, Roginsky AL (2019*) Transitioning the Use of Cryptographic Algorithms and Key Lengt*hs. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-131A, Rev. 2. https://doi.org/10.6028/NIST.SP.800-131Ar2

[WS19] Woodage J, Shumow D (2019) An Analysis of NIST SP 800-90A. In: Ishai Y, Rijmen V (eds) Advances in Cryptology – EUROCRYPT 2019. EUROCRYPT 2019. Lecture Notes in Computer Science, vol 11477. Springer, Cham. https://doi.org/10.1007/978-3-030-17656-3_6

Appendix A. Entropy vs. Security Strength (Informative)

This section of the appendix compares and contrasts entropy and security strength.

A.1. Entropy

Suppose that an entropy source produces n-bit strings with m bits of entropy in each bitstring. This means that when an n-bit string is obtained from that entropy source, the best possible guess of the value of the string has a probability of no more than 2^{-m} of being correct.

Entropy can be thought of as a property of a probability distribution, like the mean or variance. Entropy measures the unpredictability or randomness of the *probability distribution on bitstrings produced by the entropy source*, not a property of any particular bitstring. However, the terminology is sometimes slightly abused by referring to a bitstring as having m bits of entropy. This simply means that the bitstring came from a source that ensures m bits of entropy in its output bitstrings.

Because of the inherent variability in the process, predicting future entropy-source outputs does not depend on an adversary's amount of computing power.

A.2. Security Strength

A deterministic cryptographic mechanism (such as one of the DRBGs defined in [SP800-90A]) has a security strength – a measure of how much computing power an adversary expects to need to defeat the security of the mechanism. If a DRBG has an s-bit security strength, an adversary who can make 2^w computations of the underlying block cipher or hash function, where $w < s$, expects to have about a 2^{w-s} probability of defeating the DRBG's security. For example, an adversary who can perform 2^{96} AES encryptions can expect to defeat the security of the CTR-DRBG that uses AES-128 with a probability of about 2^{-32} (i.e., 2^{96-128}).

A.3. A Side-by-Side Comparison

Informally, one way of thinking of the difference between security strength and entropy is the following: suppose that an adversary somehow obtains the internal state of an entropy source (e.g., the state of all of the ring oscillators and any internal buffer). This might allow the adversary to predict the next few bits from the entropy source (assuming that there is some buffering of bits within the entropy source), but the entropy source outputs will once more become unpredictable to the adversary very quickly. For example, knowing what faces of the dice are showing on the craps table does not allow a player to successfully predict the next roll of the dice.

In contrast, suppose that an adversary somehow obtains the internal state of a DRBG. Because the DRBG is deterministic, the adversary can then predict all future outputs from the DRBG until the next reseeding of the DRBG with a sufficient amount of entropy.

An entropy source provides bitstrings that are hard for an adversary to guess correctly but usually have some detectable statistical flaws (e.g., they may have slightly biased bits, or successive bits may be correlated). However, a well-designed DRBG provides bitstrings that exhibit none of these properties. Rather, they have independent and identically distributed bits, with each bit taking on a value with a probability of exactly 0.5. These bitstrings are only unpredictable to an adversary who does not know the DRBG's internal state.

A.4. Entropy and Security Strength in this Recommendation

In the RBG1 construction specified in Section 4, the DRBG is instantiated from either an RBG2(P) or an RBG3 construction. In order to instantiate the RBG1 construction at a security strength of *s* bits, this Recommendation requires the source RBG to support a security strength of at least *s* bits and provide a bitstring that is $3s/2$ bits long for most of the DRBGs. However, for a CTR_DRBG without a derivation function, a bitstring that is $s + 128$ bits long is required. (Note that an RBG3 construction supports any desired security strength.)

In the RBG2 and RBG3 constructions specified in Sections 5 and 6, respectively, the DRBG within the construction is instantiated using a bitstring with a certain amount of entropy obtained from a validated entropy source.[34] In order to instantiate the DRBG to support an *s*-bit security strength, a bitstring with at least $3s/2$ bits of entropy is required for the instantiation of most of the DRBGs. Reseeding requires a bitstring with at least *s* bits of entropy. However, for a CTR_DRBG without a derivation function, a bitstring with exactly $s + 128$ full-entropy bits is required for instantiation and reseeding, either obtained directly from an entropy source that provides full-entropy output or from an entropy source via an **approved** (vetted) conditioning function (see Section 3.3).

The RBG3 constructions specified in Section 6 are designed to provide full-entropy outputs but with a DRBG included in the design in case the entropy source fails undetectably. Entropy bits are possibly obtained from an entropy source via an **approved** (vetted) conditioning function. When the entropy source is working properly, an *n*-bit output from the RBG3 construction is said to provide *n* bits of entropy. The DRBG in an RBG3 construction is always required to support a 256-bit security strength. If an entropy-source fails and the failure is undetected, the RBG3 construction outputs are generated at a security strength of 256 bits. In this case, the security strength of a bitstring produced by the RBG is the minimum of 256 and its length (i.e., *security_strength* = **min**(256, *length*)).

In conclusion, entropy sources and properly functioning RBG3 constructions provide output with entropy. RBG1 and RBG2 constructions provide output with a security strength that depends on the security strength of the RBG instantiation and the length of the output. Likewise, if the entropy source used by an RBG3 construction fails undetectably, the output is then dependent on the DRBG within the construction (an RBG(P) construction) to produce output at a security strength of 256 bits.

Because of the difference between the use of "entropy" to describe the output of an entropy source and the use of "security strength" to describe the output of a DRBG, the term "randomness" is used as a general term to mean either "entropy" or "security strength," as appropriate. A "randomness source" is the general term for an entropy source or RBG that provides the randomness used by an RBG.

[34] However, note that the entropy-source output may be cryptographically processed by an **approved** conditioning function before being used.

Appendix B. RBG Examples (Informative)

Appendix B.1 discusses and provides an example of the direct access to a DRBG used by an RBG3 construction.

Appendices B.2 – B.6 provide examples of each RBG construction. Not shown in the figures: if an error that indicates an RBG failure (e.g., a noise source in the entropy source has failed) is reported, RBG operation is terminated (see Section 7.1.2). For these examples, all entropy sources are considered to be physical entropy sources.

B.1. Direct DRBG Access in an RBG3 Construction

An implementation may be designed so that the DRBG implementation used within an RBG3 construction can be directly accessed by a consuming application[35] using the same or separate instantiations from the instantiation used by the RBG3 construction (see the examples in Figure 19).

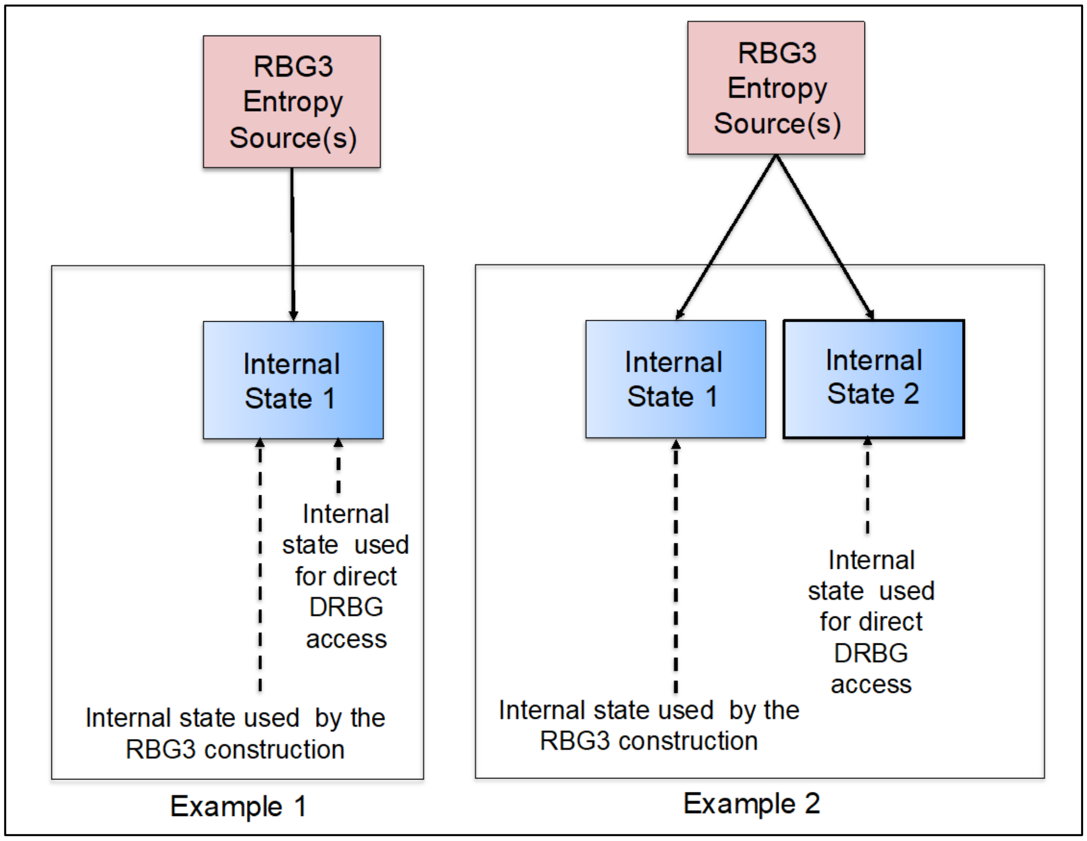
RBG3 Entropy Source(s)
Internal State 1
Internal state used for direct DRBG access
Internal state used by the RBG3 construction
Example 1
RBG3 Entropy Source(s)
Internal State 1
Internal State 2
Internal state used for direct DRBG access
Internal state used by the RBG3 construction
Example 2

Fig. 19. DRBG Instantiations

In the leftmost example in Figure 19, the same internal state is used by the RBG3 construction and a directly accessible DRBG. The DRBG implementation is instantiated only once, and only a single state handle is obtained during instantiation (e.g., *RBG3_state handle*).[36] Generation and

[35] Without using other components or functionality used by the RBG3 construction (see Sections 6.2 and 6.3).
[36] Because only a single instantiation has been implemented, a state handle is not required.

reseeding for RBG3 operations use RBG3 function calls (see Sections 6.2 and 6.3), while generation and reseeding for direct DRBG access use RBG2 function calls (see Section 5.2) with the *RBG3_state_handle*. Using the same instantiation for both RBG3 operation and direct access to the DRBG requires additional reseeding processes in the case of an RBG3(RS) construction (see Section 6.3.2).

In the rightmost example in Figure 19, different internal states are used by the RBG3 construction and a directly accessible DRBG. The DRBG implementation is instantiated twice – once for RBG3 operations and a second time for direct access to the DRBG. A different state handle needs to be obtained for each instantiation (e.g., *RBG3_state_handle* and *DRBG_state_handl*e). Generation and reseeding for RBG3 operations use RBG3 function calls and *RBG3_state_handle* (see Sections 6.2 and 6.3), while generation and reseeding for direct DRBG access use RBG2 function calls and *DRBG_state_handle* (see Section 5.2).

Multiple directly accessible DRBGs may also be incorporated into an implementation by creating multiple instantiations. However, no more than one directly accessible DRBG should share the same internal state with the RBG3 construction (i.e., if *n* directly accessible DRBGs are required, either *n* or *n*– 1 separate instantiations are required).

The directly accessed DRBG instantiations are in the same security boundary as the RBG3 construction. When accessed directly (rather than operating as part of the RBG3 construction), the DRBG instantiations are considered to be operating as RBG2(P) constructions as discussed in Section 5.

B.2. Example of an RBG1 Construction

An RBG1 construction has access to a randomness source only during instantiation when it is seeded (see Section 4). For this example (see Figure 20), the DRBG used by the RBG1 construction and the randomness source reside in two different cryptographic modules with a secure channel connecting them during the instantiation process. Following DRBG instantiation, the secure channel is not available. For this example, the randomness source is an RBG2(P) construction (see Section 5) with a state handle of *RBG2_state_handle*.

The targeted security strength for the RBG1 construction is 256 bits, so a DRBG from [SP800-90A] that is able to support this security strength must be used (HMAC_DRBG using SHA-256 is used in this example). A *personalization_string* is provided during instantiation, as recommended in Section 2.4.1.

As discussed in Section 4, the randomness source (i.e., the RBG2(P) construction for this example) is not available during normal operation, so reseeding and prediction resistance cannot be provided.

This example provides an RBG that is instantiated at a security strength of 256 bits.

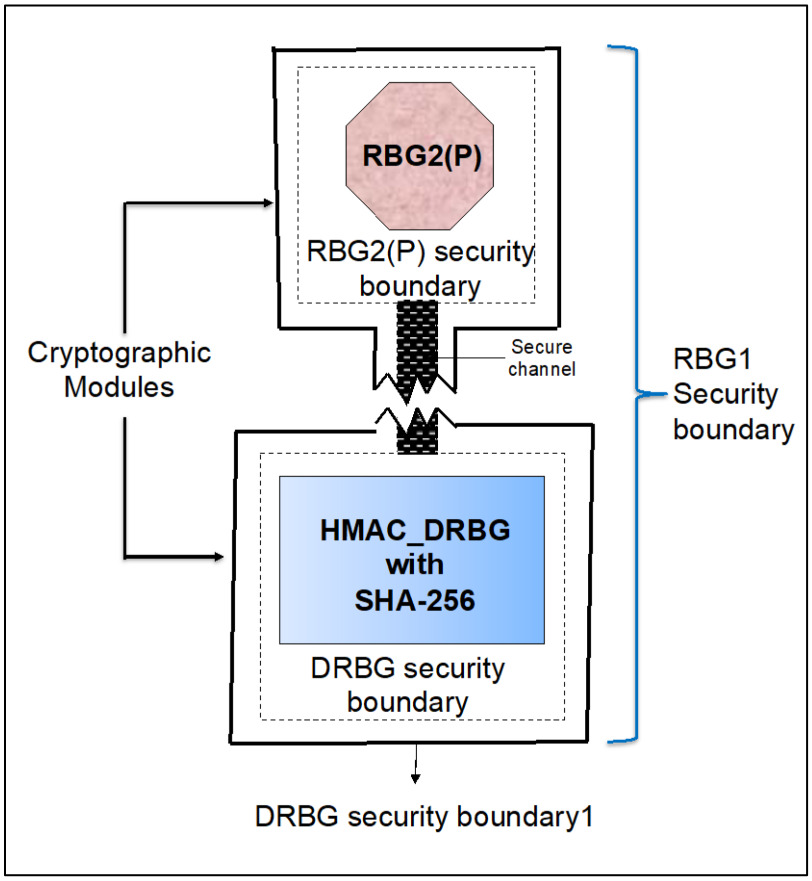

Fig. 20. RBG1 Construction Example

B.2.1. Instantiation of the RBG1 Construction

A physically secure channel is required to transport the entropy bits from the randomness source (the RBG2(P) construction) to the HMAC_DRBG during instantiation; an example of an RBG2(P) construction is provided in Appendix B.4. Thereafter, the randomness source and the secure channel are no longer available.

The HMAC_DRBG is instantiated using the **Instantiate_function**, as specified in Section 2.8.1.1, with the following call:

(*status*, *RBG1_state_handle*) = **Instantiate_function** (256, *prediction_resistance_flag* = FALSE, "Device 7056").

A security strength of 256 bits is requested for the HMAC_DRBG used in the RBG1 construction.

Since an RBG1 construction does not provide prediction resistance (see Section 4), the *prediction_resistance_flag* is set to FALSE.

The *personalization string* to be used for this example is "Device 7056."

The **Get_randomness-source_input** call in the **Instantiate_function** results in a single request being sent to the randomness source to generate bits to establish the security strength (see Section 4.2.1, item 2.a).

The HMAC_DRBG requests 3*s*/2 = 384 bits from the randomness source, where *s* = the 256-bit targeted security strength for the DRBG:

(*status*, *randomness_bitstring*) = **Generate_function**(*RBG2_state_handle*, 384, 256, *prediction_resistance_request* = TRUE).

This call requests the randomness source (indicated by *RBG2_state_handle*) to generate 384 bits at a security strength of 256 bits for the randomness input required for seeding the DRBG in the RBG1 construction. Prediction resistance is requested so that the randomness source (i.e., the RBG2(P) construction) is reseeded before generating the requested 384 bits (see Requirement 17 in Section 4.4.1). Note that optional *additional_input* is not provided for this example.

2. The RBG2(P) construction checks that the request can be handled (e.g., whether a security strength of 256 bits is supported). If the request is valid, 384 bits are generated after reseeding the RBG2(P) construction, the internal state of the RBG2(P) construction is updated, and *status* = SUCCESS is returned to the RBG1 construction along with the newly generated *randomness_bitstring*.

   If the request is determined to be invalid, *status* = FAILURE is returned along with a *Null* bitstring as the *randomnessy_bitstring*. The FAILURE *status* is subsequently returned from the **Instantiate_function** along with a Null value as the *RBG1_state_handle*, and the instantiation process is terminated.

If a valid *randomness_bitstring* is returned from the RBG2(P) construction, the *randomness_bitstring* is used along with the *personalization_string* to create the seed to instantiate the DRBG (see [SP800-90A]).[37] If the instantiation is successful, the internal state is established, a *status* of SUCCESS is returned from the **Instantiate_function** with a state handle of *RBG1_state_handle*, and the RBG can be used to generate pseudorandom bits.

B.2.2. Generation by the RBG1 Construction

Assuming that the HMAC_DRBG in the RBG1 construction has been instantiated (see Appendix B.2.1), pseudorandom bits are requested from the RBG by a consuming application using the **Generate_function** call as specified in Section 2.8.1.2:

(*status*, *returned_bits*) = **Generate_function** (*RBG1_state_handle*, *requested_number_of_bits, requested_security_strength, prediction_resistance_request* = FALSE*, additional_input*).

RBG1_state_handle was returned as the state handle during instantiation (see Appendix B.2.1).

[37] The first 256 bits of the *randomness_bitstring* are used as the randomness input, and the remaining 128 bits are used as the nonce in SP 800-90A, Revision 1. A future update of SP 800-90A will revise this process by using the entire 384-bit string as the randomness input.

The *requested_security_strength* may be any value that is less than or equal to 256 (the instantiated security strength recorded in the DRBG's internal state).

Since prediction resistance cannot be provided in an RBG1 construction, *prediction_resistance_request* is set to FALSE. (Note that the *prediction_resistance request* input parameter could be omitted from the **Generate_function** call for this example).

Any *additional input* is optional.

The **Generate_function** returns an indication of the *status*. If *status* = SUCCESS, the *requested_number_of_bits* are provided as the *returned_bits* to the consuming application. If *status* = FAILURE, *returned_bits* is an empty (i.e., null) bitstring.

B.3. Example Using Sub-DRBGs Based on an RBG1 Construction

This example uses an RBG1 construction to instantiate two sub-DRBGs: sub-DRBG1 and sub-DRBG2 (see Figure 21).

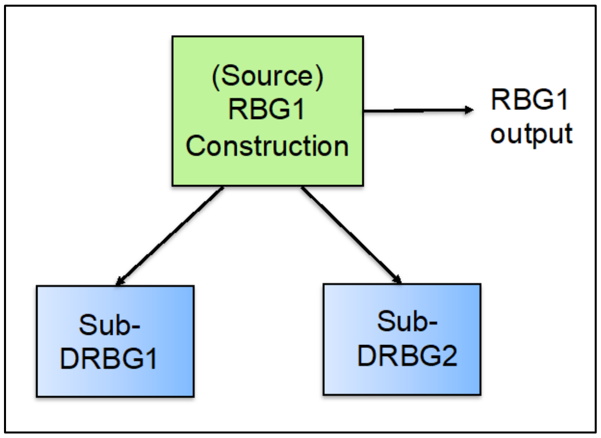

Fig. 21. Sub-DRBGs Based on an RBG1 Construction

The instantiation of the RBG1 construction is discussed in Appendix B.2. The RBG1 construction that is used as the source RBG includes an HMAC_DRBG and has been instantiated to provide a security strength of 256 bits. The state handle for the construction is *RBG1_state_handle*.

For this example, Sub-DRBG1 will be instantiated to provide a security strength of 128 bits, and Sub-DRBG2 will be instantiated to provide a security strength of 256 bits. Both sub-DRBGs use the same DRBG algorithm as the RBG1 construction.

Neither the RBG1 construction nor the sub-DRBGs can be reseeded or provide prediction resistance.

This example provides the following capabilities:

- Access to the RBG1 construction to provide output generated at a security strength of 256 bits (see Appendix B.2 for the RBG1 example)
- Access to one sub-DRBG (Sub-DRBG1) that provides output for an application that requires a security strength of no more than 128 bits

- Access to a second sub-DRBG (Sub-DRBG2) that provides output for a second application that requires a security strength of 256 bits

B.3.1. Instantiation of the Sub-DRBGs

Each sub-DRBG is instantiated using output from an RBG1 construction that is discussed in Appendix 62B.2.

B.3.1.1. Instantiating Sub-DRBG1

Sub-DRBG1 is instantiated using the following **Instantiate_function** call (see Section 2.8.1.1):

(*status*, *sub-DRBG1_state_handle*) = **Instantiate_function** (128, *prediction_resistance_flag* = FALSE, "Sub-DRBG App 1").

- A security strength of 128 bits is requested from the DRBG indicated by the *RBG1_state_handle*.
- Setting "*prediction_resistance_flag* = FALSE" indicates that a consuming application will not be allowed to request prediction resistance. Optionally, the parameter can be omitted.
- The *personalization string* to be used for sub-DRBG1 is "Sub-DRBG App 1."
- The returned state handle for sub-DRBG1 will be *sub-DRBG1_state_handle.*

The randomness input for establishing the 128-bit security strength of sub-DRBG1 is requested using the following **Generate_function** call to the RBG1 construction):

(*status*, *randomness-source_input*) = **Generate_function**(*RBG1_state_handle*, 192, 128, *prediction_resistance_request* = FALSE, *additional_input*).

- 192 bits are requested from the source RBG (indicated by *RBG1_state_handle*) at a security strength of 128 bits (192 = 128 + 64 = 3*s*/2).
- Setting "*prediction_resistance_flag* = FALSE" indicates that the source RBG (the RBG1 construction) will not need to reseed itself before generating the requested output. Alternatively, the parameter can be omitted.
- Additional input is optional.

If *status* = SUCCESS is returned from the **Generate_function**, the HMAC_DRBG in sub-DRBG1 is seeded using the *randomness-source_input* obtained from the RBG1 construction and the *personalization_string* provided in the **Instantiate_function call** (i.e., "Sub-DRBG App 1"). The internal state is recorded for Sub-DRBG1 (including the 128-bit security strength), and *status* = SUCCESS is returned from the **Instantiate_function** along with a state handle of *sub-DRBG1_state_handle*.

If *status* = FAILURE is returned from the **Generate_function** call, then the internal state is not created, *status* = FAILURE and a Null state handle are returned from the **Instantiate_function**, and the sub-DRBG1 cannot be used to generate bits.

B.3.1.2. Instantiating Sub-DRBG2

Sub-DRBG2 is instantiated using the following **Instantiate_function** call (see Section 2.8.1.1):

(*status*, *sub-DRBG2_state_handle*) = **Instantiate_function** (256, *prediction_resistance_flag* = FALSE, "Sub-DRBG App 2").

- A security strength of 256 bits is requested from the randomness source (the DRBG construction indicated by *RBG1_state_handle*).
- Setting "*prediction_resistance_flag* = FALSE" indicates that a consuming application will not be allowed to request prediction resistance. Optionally, the parameter can be omitted.
- The *personalization string* to be used for sub-DRBG2 is "Sub-DRBG App 2."
- The returned state handle will be *sub-DRBG2_state_handle.*

The randomness input for establishing the 256-bit security strength of sub-DRBG2 is requested using the following **Generate_function** call to the RBG1 construction):

(*status*, *randomness-source_input*) = **Generate_function**(*RBG1_state_handle*, 384, 256, *prediction_resistance_request* = FALSE, *additional_input*).

- 384 bits are requested from the source RBG (indicated by *RBG1_state_handle*) at a security strength of 256 bits (384 = 256 + 128 = 3*s*/2).
- Setting "*prediction_resistance_flag* = FALSE" indicates that the source RBG (the RBG1 construction) will not need to reseed itself before generating the requested output. Alternatively, the parameter can be omitted.
- Additional input is optional.

If *status* = SUCCESS is returned from the **Generate_function**, the HMAC_DRBG in sub-DRBG2 is seeded using the *randomness-source_input* obtained from the RBG1 construction and the *personalization_string* provided in the **Instantiate_function call** (i.e., "Sub-DRBG App 2"). The internal state is recorded for Sub-DRBG2 (including the 256-bit security strength), and *status* = SUCCESS is returned from the **Instantiate_function** along with a state handle of *sub-DRBG2_state_handle*.

If *status* = FAILURE is returned from the **Generate_function** call, then the internal state is not created, *status* = FAILURE and a Null state handle are returned from the **Instantiate_function**, and the sub-DRBG2 cannot be used to generate bits.

B.3.2. Pseudorandom Bit Generation by Sub-DRBGs

Assuming that the sub-DRBG has been successfully instantiated (see Appendix B.3.1), pseudorandom bits are requested from the sub-DRBG by a consuming application using the **Generate_function** call as specified in Section 2.8.1.2:

(*status*, *returned_bits*) = **Generate_function**(*state_handle*, *requested_number_of_bits*, *security_strength*, *prediction_resistance_request*, *additional input*),

where:

- For sub_DRBG1, *state_handle* = *sub-DRBG1_state_handle*;

For sub-DRBG2, *state_handle* = *sub-DRBG2_state_handle*;

- *requested_number_of_bits* must be ≤ 2^{19} (see SP 800-90A for HMAC_DRBG);
- For *sub_DRBG1, security strength* must be ≤ 128;
- For *sub_DRBG2, security strength* must be ≤ 256;
- *prediction_resistance_request* = FALSE (or is omitted); and
- *additional_input* is optional.

B.4. Example of an RBG2(P) or RBG2(NP) Construction

For this example of an RBG2 construction, no conditioning function is used, and only a single DRBG instantiation will be used (see Figure 22), so a state handle is not needed. Full-entropy output is not provided by the entropy source, which may be either a physical or non-physical entropy source.

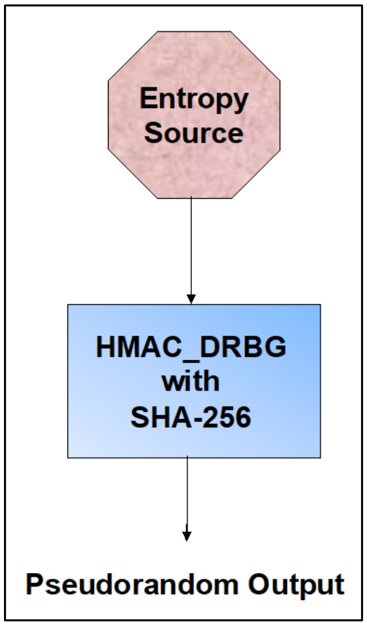

Fig. 22. RBG2 Example

The targeted security strength is 256 bits, so a DRBG from [SP800-90A] that can support this security strength must be used; HMAC_DRBG using SHA-256 is used in this example. A *personalization_string* may be provided, as recommended in Section 2.4.1. Reseeding and prediction resistance are supported and will be available on demand.

This example provides the following capabilities:

- An RBG instantiated at a security strength of 256 bits, and
- Access to an entropy source to provide prediction resistance.

B.4.1. Instantiation of an RBG2 Construction

The DRBG in the RBG2 construction is instantiated using an **Instantiate_function** call (see Section 2.8.1.1):

(*status*) = **Instantiate_function** (256, *prediction_resistance_flag* = TRUE, "RBG2 42").

- Since there is only a single instantiation, a *state_handle* is not used for this example.
- Using "*prediction_resistance_flag* = TRUE", the RBG is notified that prediction resistance may be requested in subsequent **Generate_function** calls.
- The *personalization string* to be used for this example is "RBG2 42."

The entropy for establishing the security strength (*s*) of the DRBG (i.e., *s* = 256 bits) is requested using the following **Get_ES_Bitstring** call to the entropy source (see Section 2.8.2.2 and item 2 in Section 5.2.1):

(*status*, *entropy_bitstring*) **= Get_ES_Bitstring**(384),

where 3*s*/2 = 384 bits of entropy are requested from the entropy source.

If *status* = SUCCESS is returned from the **Get_ES_Bitstring** call, the HMAC_DRBG is seeded using *entropy_bitstring*, and the *personalization_string* is "RBG2 42." The internal state is recorded (including the security strength of the instantiation), and *status* = SUCCESS is returned to the consuming application by the **Instantiate_function**.

If *status* = FAILURE is returned from the **Get_ES_Bitstring** call, then the internal state is <u>not</u> created, *status* = FAILURE and a Null state handle are returned by the **Instantiate_function** to the consuming application, and the RBG <u>cannot</u> be used to generate bits.

B.4.2. Generation in an RBG2 Construction

Assuming that the RBG has been successfully instantiated (see Appendix B.4.1), pseudorandom bits are requested from the RBG by a consuming application using the **Generate_function** call as specified in Section 2.8.1.2:

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

- Since there is only a single instantiation of the HMAC_DRBG, a *state_handle* was not returned from the **Instantiate_function** (see Appendix B.4.1) and is not used during the **Generate_function** call.
- The *requested_security_strength* may be any value that is less than or equal to 256 (the instantiated security strength recorded in the HMAC_DRBG's internal state).
- *prediction_resistance_request* = TRUE if prediction resistance is requested and FALSE otherwise.
- Additional input is optional.

If prediction resistance is requested, a reseed of the HMAC_DRBG is requested by the **Generate_function** before the requested bits are generated (see Appendix B.4). If *status* =

FAILURE is returned from the **Reseed_function**, *status* = FAILURE is also returned to the consuming application by the **Generate_function**, along with a Null value as the *returned_bits*.

Whether or not prediction resistance is requested, a *status* indication is returned from the **Generate_function** call. If *status* = SUCCESS, a bitstring of at least *requested_number_of_bits* is provided as the *returned_bits* to the consuming application. If *status* = FAILURE, *returned_bits* is an empty bitstring.

B.4.3. Reseeding an RBG2 Construction

The HMAC_DRBG will be reseeded 1) if explicitly requested by the consuming application, 2) whenever generation with prediction resistance is requested by the **Generate_function**, or 3) automatically during a **Generate_function** call at the end of the DRBG's designed *seedlife* (see the **Generate_function** specification in [SP800-90A]).

The **Reseed_function** call, as specified in Section 2.8.1.3, is:

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

- Since there is only a single instantiation of the HMAC_DRBG, a *state_handle* was not returned from the **Instantiate_function** (see Appendix B.4.1) and is not used during the **Reseed_function** call.
- The *additional_input* is optional.

Since entropy is obtained directly from the entropy source (case 2 in Section 5.2.3), the implementation has replaced the **Get_randomness-source_input** call used by the **Reseed_function** in [SP800-90A] with a **Get_ES_Bitstring** call.

The HMAC_DRBG is reseeded with a security strength of 256 bits as follows:

(*status*, *entropy_bitstring*) = **Get_ES_Bitstring**(256).

If *status* = SUCCESS is returned by **Get_ES_Bitstring**, the *entropy_bitstring* contains at least 256 bits of entropy and is at least 256 bits long. *Status* = SUCCESS is returned to the calling application (e.g., the **Generate_function**) by the **Reseed_function**.

If *status* = FAILURE, *entropy_bitstring* is an empty (e.g., null) bitstring. The HMAC_DRBG is not reseeded, and *status*= FAILURE is returned from **Reseed_function** to the calling application.

B.5. Example of an RBG3(XOR) Construction

This construction is specified in Section 6.2 and requires a DRBG and a source of full-entropy bits. For this example, the entropy source itself does not provide full-entropy output, so the vetted Hash conditioning function listed in [SP800-90B] using SHA-256 is used as an external conditioning function.

The Hash_DRBG specified in [SP800-90A] will be used as the DRBG, with SHA-256 used as the underlying hash function for the DRBG (note the use of SHA-256 for both the Hash_DRBG and the vetted conditioning function). The DRBG will obtain input directly from the RBG's entropy source without conditioning (as shown in Figure 23), since bits with full entropy are not required for input to the DRBG, even though full-entropy bits are required for input to the XOR operation (shown as "⊕" in the figure) from the entropy source via the conditioning function.

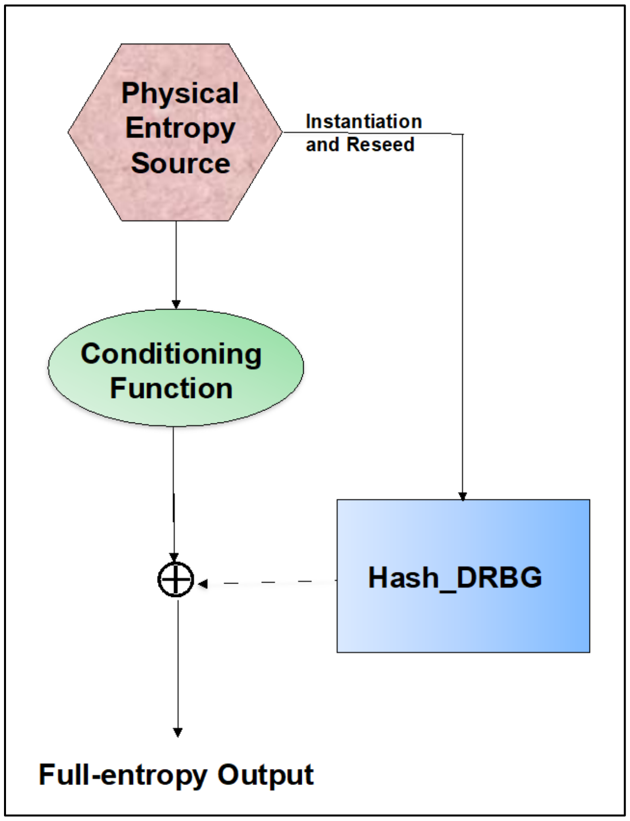

Fig. 23. RBG3(XOR) Construction Example

As specified in Section 6.2, the DRBG must be instantiated (and reseeded) at 256 bits, which is possible for SHA-256.

In this example, only a single instantiation is used, and a personalization string is provided during instantiation. The DRBG is not directly accessible.

Calls are made to the RBG using the RBG3(XOR) calls specified in Section 6.2.

The Hash_DRBG itself is not directly accessible.

This example provides the following capabilities:

- Full-entropy output by the RBG,
- Fallback to the security strength provided by the Hash_DRBG (256 bits) if the entropy source has an undetected failure, and
- Access to an entropy source to instantiate and reseed the Hash_DRBG.

B.5.1. Instantiation of an RBG3(XOR) Construction

The Hash_DRBG is instantiated using:

status = **RBG3(XOR)_DRBG_Instantiate**("RBG3(XOR)"),

- Since the DRBG is not directly accessible, there is no need for a separate instantiation, so there is also no need for the return of a state handle.
- The personalization string for the DRBG is "RBG3(XOR)."

The **RBG3(XOR)_DRBG_Instantiate** function in Section 6.2.1.1 uses a DRBG **Instantiate_function** to seed the Hash_DRBG:

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

- Since the DRBG is not directly accessible, there is no need for a separate instantiation, so there is also no need for the return of a state handle.
- The DRBG is instantiated at a security strength of 256 bits.
- The DRBG is notified that prediction resistance is not required using *prediction_resistance_flag* = FALSE. Since the DRBG will not be accessed directly, *prediction_resistance* will never be requested. Optionally, the implementation could omit this parameter.
- The personalization string for the DRBG is "RBG3(XOR)." It was provided in the **RBG3(XOR)_DRBG_Instantiate** call.

Section 6.2.1.1 refers to Section 5.2.1 for further information on instantiating the DRBG.

The entropy for establishing the security strength (*s*) of the Hash_DRBG (i.e., where *s* = 256 bits) is requested using the following **Get_ES_Bitstring** call:

(*status*, *entropy_bitstring*) **= Get_ES_Bitstring**(384),

where 3*s*/2 = 384 bits of entropy are requested from the entropy source.

If *status* = SUCCESS is returned from the **Get_ES_Bitstring** call, the Hash_DRBG is seeded using the *entropy_ bitstring* and the *personalization_string* ("RBG3(XOR)"). The internal state is recorded (including the 256-bit security strength of the instantiation), and *status* = SUCCESS is returned to the consuming application by the **Instantiate_function**. The RBG can be used to generate full-entropy bits.

If *status* = FAILURE is returned from the **Get_ES_Bitstring** call, *status* = FAILURE and a Null state handle are returned to the consuming application from the **Instantiate_function**. `The Hash_DRBG's internal state is not established, and the RBG cannot be used to generate bits.

B.5.2. Generation by an RBG3(XOR) Construction

Assuming that the Hash_DRBG has been instantiated (see Appendix B.4.1), the RBG can be called by a consuming application to generate output with full entropy.

B.5.2.1. Generation

Let *n* indicate the requested number of bits to generate. The construction in Section 6.3.1.2 is used as follows:

RBG3(XOR)_Generate:

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

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

Process:

```
1. (status, ES_bits) = Get_conditioned_full-entropy_input(n).
2. If (status ≠ SUCCESS), then return(status, Null).
3. (status, DRBG_bits) = Generate_function(n, 256, prediction_resistance_request =
   FALSE, additional_input).
4. If (status ≠ SUCCESS), then return(status, Null).
5. returned_bits = ES_bits ⊕ DRBG_bits.
6. Return SUCCESS, returned_bits.
```

Note that the *state_handle* parameter is not used in the **RBG3(XOR)_Generate** call or the **Generate_function** call (in step 3) for this example since a *state_handle* was not returned from the **RBG3(XOR)_DRBG_Instantiate** function (see Appendix B.5.1).

In step 1, the entropy source is accessed via the conditioning function using the **Get_conditioned_full-entropy_input** routine (see Appendix B.5.2.2) to obtain *n* bits with full entropy.

Step 2 checks that the **Get_conditioned_full-entropy_input** call in step 1 was successful. If it was not successful, the **RBG3(XOR)_Generate** function is aborted, returning *status* ≠ SUCCESS to the consuming application along with a *Null* bitstring as the *returned_bits*.

Step 3 calls the Hash_DRBG to generate *n* bits to be XORed with the *n*-bit output of the entropy source (*ES_Bits*; see step 1) in order to produce the RBG output. Note that a request for prediction resistance is not made in the **Generate_function** call (i.e., *prediction_resistance_request* = FALSE). Optionally, this parameter could be omitted since prediction resistance is never requested.

Step 4 checks that the **Generate_function** invoked in step 3 was successful. If it was not successful, the **RBG3(XOR)_Generate** function is aborted, returning *status* ≠ SUCCESS to the consuming application along with a *Null* bitstring as the *returned_bits*.

If step 3 returns an indication of success, the *ES_bits* returned in step 1 and the *DRBG_bits* obtained in step 3 are XORed together in step 5. The result is returned to the consuming application in step 6.

B.5.2.2. Get_conditioned_full-entropy_input Function

The **Get_conditioned_full-entropy_input** construction is specified in Section 3.3.2. For this example, the routine becomes the following:

```
Get_conditioned_full_entropy_ input:
    Input: integer n.
    Output: integer status, bitstring Full-entropy_bitstring.
Process:
    1. temp = the Null string.
    2. ctr = 0.
    3. While ctr < n, do
        3.1 (status, entropy_bitstring) = Get_ES_Bitstring (320).
        3.2 If (status ≠ SUCCESS), then return (status, invalid_string).
        3.3 conditioned_output = Hash_SHA_256(entropy_bitstring).
        3.4 temp =temp || conditioned_output.
        3.5 ctr = ctr + 256.
    4. Full-entropy_bitstring = leftmost(temp, n).
    5. Return (SUCCESS, Full-entropy_bitstring).
```

Steps 1 and 2 initialize the temporary bitstring (*temp*) for holding the full-entropy bitstring being assembled, and the counter (*ctr*) that counts the number of full-entropy bits produced so far.

Step 3 obtains and processes the entropy for each iteration.

- Step 3.1 requests 320 bits from the entropy source(s) (i.e., *output_len* + 64 bits, where *output_len* = 256 for SHA-256).
- Step 3.2 checks whether or not the *status* returned in step 3.1 indicated a success. If the *status* did not indicate a success, the *status* is returned along with an invalid (e.g., *Null*) bitstring as the *Full-entropy_bitstring*.
- Step 3.3 invokes the Hash conditioning function (see Section 3.3.1.2) using SHA-256 for processing the *entropy_bitstring* obtained from step 3.1.
- Step 3.4 concatenates the *conditioned_output* received in step 3.3 to the temporary bitstring (*temp*), and step 3.5 increments the counter for the number of full-entropy bits that have been produced so far.

After at least *n* bits have been produced in step 3, step 4 selects the leftmost *n* bits of the temporary string (*temp*) to be returned as the bitstring with full entropy.

Step 5 returns the result from step 4 (*Full-entropy_bitstring*).

B.5.3. Reseeding an RBG3(XOR) Construction

The Hash_DRBG must be reseeded at the end of its designed seedlife and may be reseeded on demand (e.g., by the consuming application). Reseeding will be automatic whenever the end of the DRBG's seedlife is reached during a **Generate_function** call (see [SP800-90A]). 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 *state_handle* parameter is not used in the **Reseed_function** call since a *state_handle* was not returned from the **RBG3(XOR)_DRBG_Instantiate** function (see Appendix B.5.1).
- The security strength for reseeding the Hash_DRBG is recorded in the internal state as 256 bits.
- Additional input is optional.

Section 6.3.1.3 refers to Section 5.2.3 for reseeding the Hash_DRBG. Since entropy is obtained directly from the entropy source and no conditioning function is used (case 2 in Section 6.3.2), the implementation has replaced the **Get_randomness-source_input** call used by the **Reseed_function** in [SP800-90A] with a **Get_ES_Bitstring** call.

The Hash_DRBG is reseeded with a security strength of 256 bits as follows:

(*status*, *entropy_bitstring*) **= Get_ES_Bitstring**(256).

If *status* = SUCCESS is returned by the **Get_ES_Bitstring** call, *entropy_bitstring* consists of at least 256 bits that contain at least 256 bits of entropy. These bits are used to reseed the Hash_DRBG. *Status* = SUCCESS is then returned to the calling application by the **Reseed_function**.

If *status* = FAILURE, *entropy_bitstring* is an empty (e.g., null) bitstring. The Hash_DRBG is not reseeded, and *status* ≠ SUCCESS is returned from the **Reseed_function** to the calling application (e.g., the **Generate_function**).

B.6. Example of an RBG3(RS) Construction

This construction is specified in Section 6.3 and requires an entropy source and a DRBG (see the left half of Figure 24 outlined in green). The DRBG is directly accessible using the same instantiation that is used by the RBG3(RS) construction (i.e., they share the same internal state). When accessed directly, the DRBG behaves as an RBG2(P) construction (see the right half of Figure 24 outlined in blue).

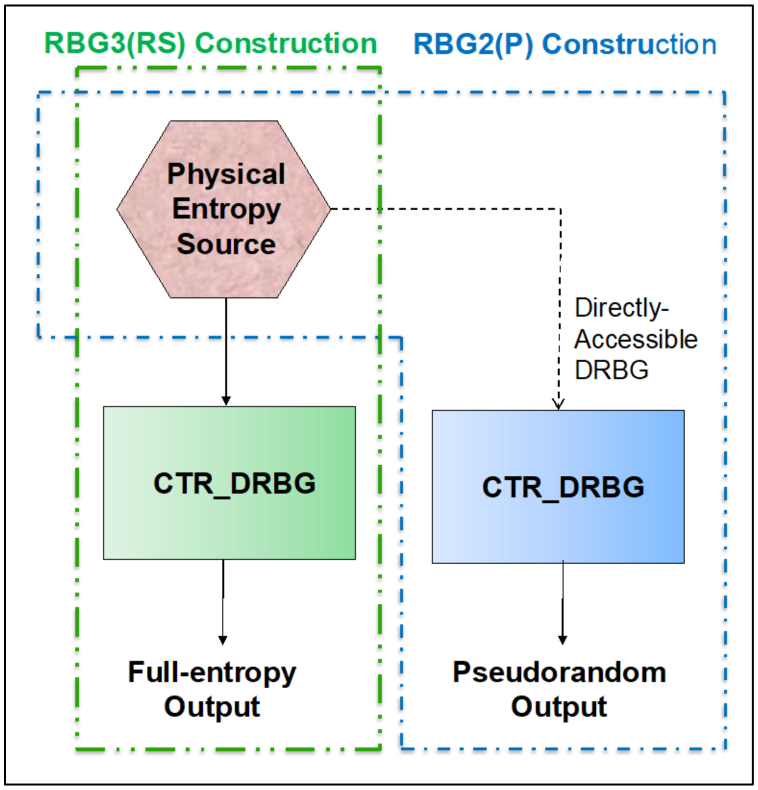

Fig. 24. RBG3(RS) Construction Example

The CTR_DRBG specified in [SP800-90A] will be used as the DRBG with AES-256 used as the underlying block cipher for the DRBG. The CTR_DRBG will be implemented using a derivation function (located inside the CTR_DRBG implementation). In this case, full-entropy output will not be required for the entropy source (see [SP800-90A]). However, an alternative example could use the CTR_DRBG without a derivation function. In that case, either the entropy source would need to provide full-entropy output, or a vetted conditioning function would be required to condition the entropy to provide full-entropy bits before providing it to the DRBG.

As specified in Section 6.2, a DRBG used as part of the RBG must be instantiated (and reseeded) at a security strength of 256 bits (which AES-256 can support).

For this example, the DRBG has a fixed security strength (256 bits), which is hard-coded into the implementation so will not be used as an input parameter.

Calls are made to the RBG as specified in Section 6.3.1. Calls made to the directly accessible DRBG (part of a RBG2(P) construction) use the RBG calls specified in Section 5.2. Since an entropy source is always available, the directly accessed DRBG can be reseeded and support prediction resistance.

If the entropy source produces output at a slow rate, a consuming application might call the RBG3(RS) construction only when full-entropy bits are required, obtaining all other output from the directly accessible DRBG.

This example provides the following capabilities:

- Full-entropy output by the RBG3(RS) construction,
- Fallback to the security strength of the RBG3(RS)'s DRBG instantiation (256 bits) if the entropy source has an undetected failure,
- Direct access to an RBG2(P) construction with a security strength of 256 bits for faster output when full-entropy output is not required,
- Access to an entropy source to instantiate and reseed the DRBG, and
- Prediction resistance support for the directly accessed DRBG.

B.6.1. Instantiation of an RBG3(RS) Construction

Instantiation for this example consists of the instantiation of the CTR_DRBG used by the RBG3(RS) construction.

The DRBG is initialized as follows:

(*status*, *RBG3(RS)_state_handle*) = **RBG3(RS)_DRBG_Instantiate**("RBG3(RS) 2021").

- "RBG3(RS) 2021" is to be used as the personalization string for the DRBG instantiation used in the RBG3(RS) construction.
- *RBG3(RS)_state_handle* is returned as the state handle for the DRBG instantiation used by the RBG3(RS) construction.

Appendices B.6.2 and B.6.3 will show the differences between the operation of the RBG3(RS) and RBG2(P) constructions.

B.6.2. Generation by an RBG3(RS) Construction

Assuming that the DRBG instantiation for the RBG3(RS) construction has been instantiated (see Appendix B.6.1), the RBG can be invoked by a consuming application to generate outputs with full entropy. The **RBG3(RS)_Generate** construction in Section 6.3.1.2.1 is invoked using

(*status*, *returned_bits*) = **RBG3(RS)_Generate**(*RBG3(RS)_state_handle*, *n,*
additional_information).

- The *RBG3(RS)_state_handle* (obtained during instantiation; see Appendix B.6.1) is used to access the internal state information for the DRBG instantiation for the RBG3(RS) construction.
- The consuming application requests *n* bits.
- The input of *additional_information* is optional.

The process is specified in Section 6.3.1.2.1. The state handle in the **Generate_function** is *RBG3(RS)_state_handle*, which was obtained during instantiation (see Appendix B.6.1).

B.6.3. Generation by the Directly Accessible DRBG

Assuming that the DRBG has been instantiated (see Appendix B.6.1), it can be accessed directly by a consuming application in the same manner as the RBG2(P) example in Appendix B.4.2 using the *RBG3(RS)_state_handle* obtained during instantiation (see Appendix B.6.1) and using a **Generate_function** call:

(*status*, *returned_bits*) = **Generate_function**(*RBG3(RS)_state_handle*, *n,*
prediction_resistance_request, additional_input).

Note that the security strength parameter (256) was omitted since its value has been hard coded.

Requirement 2 in Section 6.3.2 requires that the DRBG be reseeded whenever a request for generation by a directly accessible DRBG follows a request for generation by the RBG3(RS) construction. For this example, the internal state includes an indication about whether the last use of the DRBG was as part of the RBG3(RS) construction or was directly accessible. If the **Generate_function** (above) does not include a request for prediction resistance (e.g., *prediction_resistance_request* was not set to TRUE), then the DRBG will be reseeded anyway using the entropy source before generating output if the previous use of the DRBG was part of the RBG3(RS) construction.

B.6.4. Reseeding a DRBG

When operating as part of the RBG3(RS) construction, the **Reseed_function** is invoked one or more times to produce full-entropy output when the **RBG3(RS)_Generate** function is invoked by a consuming application.

When operating as part of the RBG2(P) construction (the directly accessible DRBG), the DRBG is 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 (see Appendix B.6.3), 3) whenever the previous use of the DRBG was by the **RBG3(RS)_Generate** function (see Appendix B.6.2), or 4) automatically during a **Generate_function** call at the end of the seedlife of the RBG2(P) construction (see the **Generate_function** specification in [SP800-90A]).

The **Reseed_function** call is:

status = **Reseed_function**(*RBG3(RS)_state_handle*, *additional_input*).

- The state_handle is *RBG3(RS)_state handle*, and
- *additional_input* is optional.[38]

The DRBG is reseeded with a security strength of 256 bits as follows:

(*status*, *entropy_bitstring*) = **Get_ES_Bitstring**(256).

If *status* = SUCCESS is returned by **Get_ES_Bitstring**, *entropy_bitstring* consists of at least 256 bits containing at least 256 bits of entropy. *Status* = SUCCESS is returned to the calling application by the **Reseed_function**.

[38] Note that when the **RBG3(RS) Generate** function uses a Hash_DRBG, HMAC_DRBG, or CTR_DRBG with no derivation function and Method A, whereby 64 bits of additional entropy are required to produce *output_len* bits with full entropy (see Section 7.3.1,.2.1, step 3.1), the additional 64 bits of entropy obtained in step 3.1.1 is provided to the **Generate_function** (in step 3.1.3) with prediction requested. In Section 9.3 of SP 800-90A, the **Generate_function** reseeds the DRBG when prediction resistance is requested using entropy from the entropy source and any additional input that is provided – the additional 64 bits of entropy, in this case.

If *status* ≠ SUCCESS (e.g., the entropy source has failed), *entropy_bitstring* is an empty (e.g., null) bitstring, the DRBG is not reseeded, and a FAILURE *status* is returned from **Reseed_function** to the calling application (e.g., the **Generate_function**).

Appendix C. Addendum to SP 800-90A: Instantiating and Reseeding a CTR_DRBG

C.1. Background and Scope

The CTR_DRBG, specified in [SP800-90A], uses the block cipher AES and has two versions that may be implemented: with or without a derivation function.

When a derivation function is not used, SP 800-90A requires the use of bitstrings with full entropy for instantiating and reseeding a CTR_DRBG. This addendum permits the use of an RBG compliant with SP 800-90C to provide the required seed material for the CTR_DRBG when implemented as specified in SP 800-90C (see Appendix C.2).

When a derivation function is used in a CTR_DRBG implementation, SP 800-90A specifies the use of the block cipher derivation function. This addendum modifies the requirements in SP 800-90A for the CTR_DRBG by specifying two additional derivation functions that may be used instead of the block cipher derivation function (see Appendix C.3).

C.2. CTR_DRBG without a Derivation Function

When a derivation function is not used, SP 800-90A requires that *seedlen* full-entropy bits be provided as the randomness input (e.g., from an entropy source that provides full-entropy output), where *seedlen* is the length of the key to be used by the CTR_DRBG plus the length of the output block.[39] SP 800-90C includes an approved method for externally conditioning the output of an entropy source to provide a bitstring with full entropy when using an entropy source that does not provide full-entropy output.

SP 800-90C also permits the use of seed material from an RBG when the DRBG to be instantiated and reseeded is implemented and used as specified in SP 800-90C.

C.3. CTR_DRBG using a Derivation Function

When a derivation function is used within a CTR_DRBG, SP 800-90A specifies the use of the **Block_cipher_df** included in that document during instantiation and reseeding to adjust the length of the seed material to *seedlen* bits, where

seedlen = the security strength + the block length.

For AES, *seedlen* = 256, 320 or 384 bits (see [SP800-90A], Rev. 1). During generation, the length of any additional input provided during the generation request is adjusted to *seedlen* bits as well (see SP 800-90A).

[39] 128 bits for AES.

Two alternative derivation functions are specified in Appendices C.3.2 and C.3.3. Appendix C.3.1 discusses the keys and constants for use with the alternative derivation functions specified in Appendices C.3.2 and C.3.3.

C.3.1. Derivation Keys and Constants

Both of the derivation methods specified in Appendices C.3.2 and C.3.3 an AES derivation key (df_Key) whose length shall meet or exceed the instantiated security strength of the DRBG instantiation.

The *df_Key* **may** be set to any value and **may** be the current value of a key used by the DRBG.

These alternative methods use three 128-bit constants C_1, C_2 and C_3, which are defined as:

C_1 = 000000...00

C_2 = 101010...10

C_3 = 010101...01

The value of B used in Appendices C.3.2 and C.3.3 depends on the length of the AES derivation key (*df_Key*). When the length of *df_Key* = 128 bits, then $B = 2$. Otherwise, $B = 3$.

C.3.2. Derivation Function Using CMAC

CMAC is a block-cipher mode of operation specified in [SP800-38B]. The CMAC_df derivation function is specified as follows:

CMAC_df:

Input: bitstring *input_string*, integer *number_of_bits_to_return.*

Output: bitstring Z.

Process:

```
1. Let $C_1$, $C_2$, $C_3$ be 128-bit blocks defined as 000000...0, 101010...10, 010101...01,
   respectively.
2. Get df_Key.     Comment: See Appendix C.3.1.
3. $Z$ = the Null string.
4. For $i = 1$ to $B$:
   $Z = Z$ || CMAC(df_Key, $C_i$ || input_string).
5. $Z$ = leftmost ($Z$, number_of_bits_to_return).
6. Return($Z$).
```

C.3.3. Derivation Function Using CBC-MAC

This CBC-MAC derivation function **shall** only be used when the *input_string* has the following properties:

- The length of the *input string* is always a fixed length.
- The length of the *input_string* is an integer multiple of 128 bits. Let *m* be the number of 128-bit blocks in the *input_string*.

This derivation function is specified as follows:

CBC-MAC_df:

Input: bitstring *input_string*, integer *number_of_bits_to_return.*

Output: bitstring *Z.*

Process:

```
1. Let $C_1$, $C_2$, $C_3$ be 128-bit blocks defined as 000000...0, 101010...10, 010101...01,
   respectively.
2. Get $df\_Key$.     Comment: See Appendix C.3.1.
3. $Z$ = the $Null$ string.
4. Let $input\_string = S_1 \| S_2 \| ... \| S_m$, where the $S_i$ are contiguous 128-bit blocks.
5. For $j = 1$ to $B$:
   5.1  $S_0 = C_j$.
   5.2  $V$ = 128-bit block of all zeroes.
   5.3  For $i = 0$ to $m$:
           $V = \text{Encrypt}(df\_Key, V \oplus S_i)$.     Comment: Perform the cipher operation
                                                        specified in [FIPS197].
   5.4  $Z = Z \| V$.
6. $Z$ = leftmost($Z$, $number\_of\_bit\_to\_return$).
7. Return($Z$).
```

Appendix D. List of Symbols, Abbreviations, and Acronyms

AES	Advanced Encryption Standard[40]
API	Application Programming Interface
CAVP	Cryptographic Algorithm Validation Program
CDF	Cumulative Distribution Function
CMVP	Cryptographic Module Validation Program
DRBG	Deterministic Random Bit Generator[41]
FIPS	Federal Information Processing Standard
ITL	Information Technology Laboratory
MAC	Message Authentication Code
NIST	National Institute of Standards and Technology
RAM	Random Access Memory
RBG	Random Bit Generator
SP	(NIST) Special Publication
Sub-DRBG	Subordinate DRBG
TDEA	Triple Data Encryption Algorithm[42]
XOR	Exclusive-Or (operation)
$\mathbf{0}^{x}$	A string of x zeroes
$\lceil \boldsymbol{x} \rceil$	

[40] As specified in [FIPS 197].
[41] Mechanism specified in [SP800-90A].
[42] As specified in [SP 800-67], Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher.

The ceiling of x; the least integer number that is not less than the real number x. For example, $\lceil 3 \rceil = 3$, and $\lceil 5.5 \rceil = 6$.

$\boldsymbol{\varepsilon}$
A positive constant that is assumed to be smaller than 2^{-32}

E(*X*)
The expected value of the random variable X

len(*x*)
The length of x in bits

min(*a*, *b*)
The minimum of a and b

output_len
The bit length of the output block of a cryptographic primitive

s
The security strength

$\boldsymbol{X \oplus Y}$
Boolean bitwise exclusive-or (also bitwise addition modulo 2) of two bitstrings X and Y of the same length

+
Addition over real numbers

×
Multiplication over real numbers

Appendix E. Glossary

adversary
A malicious entity whose goal is to determine, to guess, or to influence the output of an RBG.

approved
An algorithm or technique for a specific cryptographic use that is specified in a FIPS or NIST Recommendation, adopted in a FIPS or NIST Recommendation, or specified in a list of NIST-approved security functions.

backtracking resistance
A property of a DRBG that provides assurance that compromising the current internal state of the DRBG does not weaken previously generated outputs. See SP 800-90A for a more complete discussion. (Contrast with *prediction resistance*.)

biased
A random variable is said to be biased if values of the finite sample space are selected with unequal probability. Contrast with unbiased.

big-endian format
A format in which 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.

bitstring
An ordered sequence (string) of 0s and 1s. The leftmost bit is the most significant bit.

block cipher
A parameterized family of permutations on bitstrings of a fixed length; the parameter that determines the permutation is a bitstring called the key.

conditioning function (external)
As used in SP 800-90C, a deterministic function that is used to produce a bitstring with full entropy.

consuming application
An application that uses random outputs from an RBG.

cryptographic boundary
An explicitly defined physical or conceptual perimeter that establishes the physical and/or logical bounds of a cryptographic module and contains all of the hardware, software, and/or firmware components of a cryptographic module.

cryptographic module
The set of hardware, software, and/or firmware that implements cryptographic functions (including cryptographic algorithms and key generation) and is contained within the cryptographic boundary.

deterministic random bit generator (DRBG)
An RBG that produces random bitstrings by applying a deterministic algorithm to initial seed material.

Note: A DRBG at least has access to a randomness source initially.

Note: A portion of the seed material is secret.

digitization
The process of generating raw discrete digital values from non-deterministic events (e.g., analog noise sources) within a noise source.

entropy
A measure of disorder, randomness, or variability in a closed system.

Note: The entropy of a random variable X is a mathematical measure of the amount of information gained by an observation of X.

Note: The most common concepts are Shannon entropy and min-entropy. Min-entropy is the measure used in SP 800-90.

entropy rate
The validated rate at which an entropy source provides entropy in terms of bits per entropy-source output (e.g., five bits of entropy per eight-bit output sample).

entropy source
The combination of a noise source, health tests, and optional conditioning component that produce bitstrings containing entropy. A distinction is made between entropy sources having physical noise sources and those having non-physical noise sources.

Note: Health tests are comprised of continuous tests and startup tests.

fresh entropy
A bitstring that is output from a non-deterministic randomness source that has not been previously used to generate output or has otherwise been made externally available.

Note: The randomness source should be an entropy source or RBG3 construction.

full-entropy bitstring
A bitstring with ideal randomness (i.e., the amount of entropy per bit is equal to 1). This Recommendation assumes that a bitstring has *full entropy* if the entropy rate is at least $1 - \varepsilon$, where ε is at most 2^{-32}.

hash function
A (mathematical) function that maps values from a large (possibly very large) domain into a smaller range. The function satisfies the following properties:

1. (One-way) It is computationally infeasible to find any input that maps to any pre-specified output.
2. (Collision-free) It is computationally infeasible to find any two distinct inputs that map to the same output.

health testing
Testing within an implementation immediately prior to or during normal operations to obtain assurance that the implementation continues to perform as implemented and validated.

ideal randomness source
The source of an ideal random sequence of bits. Each bit of an ideal random sequence is unpredictable and unbiased, with a value that is independent of the values of the other bits in the sequence. Prior to an 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
Two entropy sources are *independent* if knowledge of the output of one entropy source provides no information about the output of the other entropy source.

instantiate
The process of initializing a DRBG with sufficient randomness to generate pseudorandom bits at the desired security strength.

internal state (of a DRBG)
The collection of all secret and non-secret information about an RBG or entropy source that is stored in memory at a given point in time.

known-answer test
A test that uses a fixed input/output pair to detect whether a deterministic component was implemented correctly or to detect whether it continues to operate correctly.

min-entropy
A lower bound on the entropy of a random variable. 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 measure of the unpredictability of a random variable.

must
Used in SP 800-90C to indicate a requirement that may not be testable by a CMVP testing lab. Note that **must** may be coupled with **not** to become **must not**.

noise source
A source of unpredictable data that outputs raw discrete digital values. The digitization mechanism is considered part of the noise source. A distinction is made between physical noise sources and non-physical noise sources.

non-physical entropy source
An entropy source whose primary noise source is non-physical.

non-physical noise source
A noise source that typically exploits system data and/or user interaction to produce digitized random data.

non-validated entropy source
An entropy source that has not been validated by the CMVP as conforming to SP 800-90B.

null string
An empty bitstring.

personalization string
An optional input value to a DRBG during instantiation to make one DRBG instantiation behave differently from other instantiations.

physical entropy source
An entropy source whose primary noise source is physical.

physical noise source
A noise source that exploits physical phenomena (e.g., thermal noise, shot noise, jitter, metastability, radioactive decay, etc.) from dedicated hardware designs (using diodes, ring oscillators, etc.) or physical experiments to produce digitized random data.

prediction resistance
A property of a DRBG that provides assurance that compromising the current internal state of the DRBG does not allow future DRBG outputs to be predicted past the point where the DRBG has been reseeded with sufficient entropy. See SP 800-90A for a more complete discussion. (Contrast with *backtracking resistance*.)

pseudocode
An informal, high-level description of a computer program, algorithm, or function that resembles a simplified programming language.

random bit generator (RBG)
A device or algorithm that outputs a random sequence that is effectively indistinguishable from statistically independent and unbiased bits.

randomness
As used in this Recommendation, the unpredictability of a bitstring. If the randomness is produced by a non-deterministic source (e.g., an entropy source or RBG3 construction), the unpredictability is dependent on the quality of the source. If

the randomness is produced by a deterministic source (e.g., a DRBG), the unpredictability is based on the capability of an adversary to break the cryptographic algorithm for producing the pseudorandom bitstring.

randomness input
An input bitstring from a randomness source that provides an assessed minimum amount of randomness (e.g., entropy) for a DRBG. See *min-entropy*.

randomness source
A source of randomness for an RBG. The randomness source may be an entropy source or an RBG construction.

RBG1 construction
An RBG construction with the DRBG and the randomness source in separate cryptographic modules.

RBG2 construction
An RBG construction with one or more entropy sources and a DRBG within the same cryptographic module. This RBG construction does not provide full-entropy output.

RBG2(NP) construction
A non-physical RBG2 construction. An RBG2 construction that obtains entropy from one or more validated non-physical entropy sources and possibly from one or more validated physical entropy sources. This RBG construction does not provide full-entropy output.

RBG2(P) construction
A physical RBG2 construction. An RBG construction that includes a DRBG and one or more entropy sources in the same cryptographic module. Only the entropy from validated physical entropy sources is counted when fulfilling an entropy request within the RBG. This RBG construction does not provide full-entropy output.

RBG3 construction
An RBG construction that includes a DRBG and one or more entropy sources in the same cryptographic module. When working properly, bitstrings that have full entropy are produced. Sometimes called a *non-deterministic random bit generator* (NRBG) or true random number (or bit) *generator*.

reseed
To refresh the internal state of a DRBG with seed material. The seed material should contain sufficient entropy to allow recovery from a possible compromise.

sample space
The set of all possible outcomes of an experiment.

secure channel
A physically protected secure path for transferring data between two cryptographic modules that ensures confidentiality, integrity, and replay protection as well as mutual authentication between the modules.

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

For a DRBG: A conceptual boundary that contains all of the DRBG functions and internal states required for a DRBG.

For an RBG: A conceptual boundary that is defined with respect to one or more threat models that includes an assessment of the applicability of an attack and the potential harm caused by the attack.

security strength
A number associated with the amount of work (i.e., 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 {128, 192, 256}. If the security strength associated with an algorithm or system is s bits, then it is expected that (roughly) 2^s basic operations are required to break it.

Note: This is a classical definition that does not consider quantum attacks. This definition will be revised to address quantum issues in the future.

seed
To initialize the internal state of a DRBG with seed material. The seed material should contain sufficient entropy to meet security requirements.

seed material
A bitstring that is used as input to a DRBG. The seed material determines a portion of the internal state of the DRBG.

seedlife
The period of time between instantiating or reseeding a DRBG with seed material and reseeding the DRBG with seed material containing fresh entropy or uninstantiation of the DRBG.

shall
The term used to indicate a requirement that is testable by a testing lab. **Shall** may be coupled with **not** to become **shall not**. See *Testable requirement*.

should
The term used to indicate an important recommendation. Ignoring the recommendation could result in undesirable results. Note that **should** may be coupled with **not** to become **should not**.

state handle
A pointer to the internal state information for a particular DRBG instantiation.

subordinate DRBG (sub-DRBG)
A DRBG that is instantiated by an RBG1 construction.

support a security strength (by a DRBG)
The DRBG has been instantiated at a security strength that is equal to or greater than the security strength requested for the generation of random bits.

targeted security strength
The security strength that is intended to be supported by one or more implementation-related choices (e.g., algorithms, cryptographic primitives, auxiliary functions, parameter sizes, and/or actual parameters).

testable requirement
A requirement that can be tested for compliance by a testing lab via operational testing, a code review, or a review of relevant documentation provided for validation. A testable requirement is indicated using a **shall** statement.

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 the potential harm from each attack.

unbiased
A random variable is said to be unbiased if all values of the finite sample space are chosen with the same probability. Contrast with biased.

uninstantiate
The termination of a DRBG instantiation.

validated entropy source
An entropy source that has been successfully validated by the CAVP and CMVP for conformance to SP 800-90B.