NIST Special Publication 800
NIST SP 800-90C 4pd

Recommendation for Random Bit Generator (RBG) Constructions

Fourth Public Draft

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.4pd

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

NIST Special Publication 800
NIST SP 800-90C 4pd

Recommendation for Random Bit Generator (RBG) Constructions

Fourth Public Draft

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.4pd

July 2024

U.S. Department of Commerce
Gina M. Raimondo, Secretary

National Institute of Standards and Technology
Laurie E. Locascio, NIST Director and Under Secretary of Commerce for Standards and Technology

Publication History
Approved by the NIST Editorial Review Board on YYYY-MM-DD [will be added in final published version]
Supersedes NIST Series XXX (Month Year) DOI [Will be added to final publication.]

How to Cite this NIST Technical Series Publication:
Barker EB, Kelsey JM, McKay KA, Roginsky AL, Sönmez Turan M (2024) Recommendation for Random Bit Generator (RBG) Constructions. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-90C 4pd. https://doi.org/10.6028/NIST.SP.800-90C.4pd

Author ORCID iDs
Elaine Barker: 000-0003-0454-0461
John Kelsey: 000-0002-3427-1744
Kerry McKay: 000-0002-5956-587X
Allen Roginsky: 0000-0003-2684-6736
Meltem Sönmez Turan: 0000-0002-1950-7130

Public Comment Period
July 3, 2024 – September 30, 2024

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

Additional Information
Additional information about this publication is available at https://csrc.nist.gov/pubs/sp/800/90/c/4pd, including related content, potential updates, and document history.

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

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 four classes of RBGs — namely, RBG1, RBG2, RBG3, and RBGC — 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; RBGC construction; subordinate DRBG (sub-DRBG).

Reports on Computer Systems Technology

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

Note to Reviewers

1. This fourth public draft of SP 800-90C describes four RBG constructions: RBG1, RBG2, RBG3, and RBGC. The RBGC construction has been included since the last draft to specify chains or trees of DRBGs. Responses to the following questions are requested for the DRBG chains discussed in Sec. 7:
   - Should the initial randomness source for the root RBGC construction be required to be “part” of the computing platform on which the DRBG chains are used (i.e., non-removable during system operation), or should an external removable device be allowed as the initial randomness source? Please provide a rationale.
   - For the DRBG tree structure in Sec. 7, will a requirement for the initial randomness source to be reseeded before generating output for seeding or reseeding the root RBGC construction be a substantial problem if that source is an RBG2(P) or RBG2(NP) construction (e.g., when dev/random is serving as the initial randomness source)? Refer to Sec. 7.1.2.1 for an example.
   - What kind of guidance should be included for virtualized and cloud environments to avoid insecure implementations?
   - Should a limit be imposed on the length of a DRBG chain? If so, what limit would be appropriate?
2. This draft distinguishes between a request for the execution of a function within a DRBG or RBG (e.g., by an application) and the execution of the requested function within the DRBG or RBG. However, note that the inputs and outputs of the request and the intended function are usually the same.
3. A prediction-resistance request in a **DRBG_Generate** function is no longer provided as an input parameter. Instead, prediction resistance can be obtained prior to issuing a generate request by first issuing a reseed request using the **DRBG_Reseed** function.
4. For an RBG2 construction (see Sec. 5), a capability for reseeding is optional. When a reseed capability is implemented, reseeding may be performed upon request by an application and/or in response to some trigger. When reseeding is supported, periodic reseeding is recommended to ensure recovery from a compromise.
   - Should a reseeding capability be required for an RBG2(P) or RBG2(NP) construction?
   - If an implementation has a reseeding capability, should reseeding be required?
   - If periodic reseeding is required, what advice should be included for reseeding an RBG2 construction? The example of reseeding after at most 2^{19} output bits is suggested to align with the requirements in AIS 20/31 in case a developer would like to submit its implementation to both the NIST and BSI validation programs.

5. SHA-1 and the 224-bit hash functions (i.e., SHA-224, SHA-512/224, and SHA3-224) have been removed from this version since NIST plans to disallow them after 2030 (see an upcoming revision of SP 800-131A).

6. After the publication of SP 800-90C, SP 800-90A (Revision 1) will be revised to resolve inconsistencies with this document. The revision will include:

   - The **Instantiate_function**, **Generate_function**, and **Reseed_function** will be renamed to **DRBG_Instantiate**, **DRBG_Generate**, and **DRBG_Reseed**. These names have been used in SP 800-90C for clarity.
   - The **Get_entropy_input** call discussed in SP 800-90Ar1 will be renamed to the more general term "**Get_randomness-source_input**," which is used in SP 800-90C.
   - SP 800-90Ar1 currently requires a nonce to be used during DRBG instantiation that is either 1) a value with at least (*security_strength*/2) bits of entropy or 2) a value that is expected to repeat no more often than a (*security_strength*/2)-bit random string would be expected to repeat. The use of the nonce (as defined in SP 800-90Ar1) will be replaced by additional bits provided by the randomness source.
   - Parameters needed to use the DRBGs in the constructions specified in SP 800-90C will be provided for each DRBG type in SP 800-90Ar1 (i.e., the Hash_DRBG, HMAC_DRBG, and CTR_DRBG).
   - Are there any other inconsistencies between this draft of SP 800-90C and the current version of SP 800-90Ar1 at https://doi.org/10.6028/NIST.SP.800-90Ar1?

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 Chris Celi and Hamilton Silberg (NIST); Darryl Buller, Aaron Kaufer, and Mike Boyle (National Security Agency); Werner Schindler, Matthias Peter, and Johannes Mittmann (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:

1. 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 randomness — can be used to generate random bits suitable for cryptographic applications.
2. SP 800-90B, *Recommendation for the Entropy Sources Used for Random Bit Generation*, provides guidance for the development and validation of entropy sources, which are 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.
3. SP 800-90C, *Recommendation for Random Bit Generator (RBG) Constructions,* specifies constructions for random bit generators (RBGs) using 1) randomness sources (either entropy sources that comply with SP 800-90B or RBGs that comply with SP 800-90C) and 2) DRBGs that comply with SP 800-90A. Four classes of RBGs are specified in this document (see Sec. 4–7). SP 800-90C also provides high-level guidance for testing RBGs for conformance to this recommendation.

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.

The RBG constructions defined in this recommendation are based on two components: the *entropy sources* that generate true random variables (i.e., variables that may be biased, where 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.

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 140-3 and the Cryptographic Module Validation Program* [FIPS_ 140IG].

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 to reduce bias and obtain full entropy when needed.
- Sections 4, 5, 6, and 7 specify the RBG constructions, namely the RBG1, RBG2, RBG3, and RBGC constructions, respectively.
- Section 8 discusses health and implementation validation testing.
- The References contain 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, generating output using the RBG3(RS) construction, and computing platforms, as required by DRBG chains using the RBGC construction.
- Appendix B provides examples of each RBG construction.
- Appendix C is an addendum for 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 no adversary 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 RBG 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 Sec. 6), which can provide full-entropy output.[1]

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}. NIST Internal Report (IR) 8427 [NISTIR _8427] provides a justification for the selection of ε.

2.2. RBG Constructions

A *construction* is a method of designing an RBG to accomplish a specific goal. Four classes of RBG constructions are defined in this document: RBG1, RBG2, RBG3, and RBGC (see Table 1). Each RBG includes a DRBG from SP 800-90A and is based on the use of a randomness source that is validated for compliance with SP 800-90B 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 Appendix A.1 for a discussion of entropy versus security strength.

Table 1. RBG capabilities

Construction	Internal Entropy Source	Available randomness source for reseeding	Prediction Resistance	Full Entropy	Type of Randomness Source
RBG1	No	No	No	No	RBG2(P) or RBG3 construction
RBG2(P)	Yes	Yes	Optional	No	Physical entropy source
RBG2(NP)	Yes	Yes	Optional	No	Non-physical entropy source
RBG3(XOR) or RBG3(RS)	Yes	Yes	Yes	Yes	Physical entropy source
(Root) RBGC	Yes	Yes	Optional	No	RBG2 or RBG3 construction or Full-entropy source
(Non-root) RBGC	No	Yes	No	No	Parent RBGC construction

In Table 1:

- Column 1 lists the RBG constructions specified in this document.
- Column 2 indicates whether an entropy source is present within the construction.
- Column 3 indicates whether the DRBG has an available randomness source for reseeding.
- Column 4 indicates whether prediction resistance can be provided for the output of the RBG (see Sec. 2.4.2 for a discussion of prediction resistance).
- Column 5 indicates whether full-entropy output can be provided by the RBG.
- Column 6 indicates the types of randomness sources that are allowed for the RBG construction.

An RBG1 construction does not have access to a randomness source after instantiation. It is instantiated once in its lifetime over a physically secure channel from an external RBG2(P) or RBG3 construction with appropriate security properties. An RBG1 construction does not support reseeding requests, prediction resistance cannot be provided for the output, and the construction cannot provide output with full entropy. The construction can be used to initialize subordinate DRBGs (sub-DRBGs) (see Sec. 4).

An RBG2 construction includes one or more entropy sources that are used to instantiate the DRBG and may (optionally) be used for reseeding if a reseed capability is implemented. Prediction resistance may be provided to the RBG output when reseeding is performed. The construction has two variants: an RBG2(P) construction uses a physical entropy source to provide entropy, while an RBG2(NP) construction uses a non-physical entropy source. An RBG2 construction cannot provide full-entropy output (see Sec. 5).

An RBG3 construction includes one or more physical entropy sources and is designed to provide an output with a security strength equal to the requested length of its output by producing outputs that have full entropy. Prediction resistance is provided for all outputs (see Sec. 6).

This construction has two types:

1. An **RBG3(XOR)** construction combines the output of one or more validated entropy sources with the output of an instantiated, approved DRBG using an exclusive-or (XOR) operation (see Sec. 6.4).
2. An **RBG3(RS)** construction uses one or more validated entropy sources to provide seed material for the DRBG by continuously reseeding.

An RBGC construction (see Sec. 7) allows the use of a chain of RBGs that consists of only RBGC constructions on the same computing platform. The initial RBGC construction in the chain is called the root RBGC construction; the root RBGC construction accesses an initial randomness source for instantiation and reseeding. Subsequent RBGC constructions in the chain are seeded (and may be reseeded) using their immediate predecessor RBGC construction (i.e., their parent). Prediction resistance may be provided for the root but not for subsequent RBGC constructions (see Sec. 6.5).

This document also provides procedures for acquiring entropy from an entropy source and conditioning the output to provide a bitstring with full entropy (see Sec. 3.2). 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 as well as text. The pseudocode conventions are not intended to constrain real-world implementations but to provide a consistent notation to describe the constructions.

For any of the specified processes, equivalent processes may be used. Two processes are equivalent if the same output is produced when the same values are input to each process (either as input parameters or as values made available during the process).

By convention and unless otherwise specified, integers are unsigned 32-bit values. 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 — mechanisms that provide entropy for an RBG. Some RBG constructions access these entropy sources directly to obtain entropy. Other constructions fulfill their entropy requirements by accessing another RBG as a randomness source, in which case the RBG used as a randomness source may include an entropy source or have a predecessor that includes an entropy source.

SP 800-90B provides guidance for the development and validation of entropy sources. Validated entropy sources (i.e., entropy sources that have been successfully validated by the CMVP as complying with SP 800-90B) reliably provide fixed-length outputs and a specified minimum amount of entropy for each output (e.g., each 8-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 within the entropy source is physical — that is, it uses a dedicated hardware design 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 within the entropy source is non-physical — that is, entropy is provided by system data (e.g., system time or the entropy present in the RAM data) or human interaction (e.g., mouse movements). The entropy source type (i.e., physical or non-physical) is certified during SP 800-90B validation.

One or more validated, independent entropy sources may be used to provide entropy for instantiating and reseeding the DRBGs in RBG2, RBG3, and (root) RBGC constructions or used by an RBG3 construction to generate output upon request by a consuming application. Appropriate validated RBGs may be used to provide seed material for RBG1 and RBGC constructions.

An implementation could be designed to use a combination of physical and non-physical entropy sources. When requests are made to these sources, bitstring outputs may be 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 independent, validated physical entropy sources; one or more validated non-physical entropy sources may also be included in the implementation. 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 independent, validated non-physical entropy sources; one or more independent, validated 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 i^{th} output of a physical entropy source and $npes_j$ be the j^{th} 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:

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

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

[2] This document also discusses the use of non-validated entropy sources. When discussing such entropy sources, "non-validated" will always precedes "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 entropy request. Any entropy in $npes_1$, ..., $npes_m$ is not counted.

According to Method 2, all the entropy in pes_1, pes_2, ..., pes_n and in $npes_1$, $npes_2$, ..., $npes_m$ is counted.

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. For example, 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 apply, providing that the entropy sources are independent.

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 contains at least the requested amount of entropy (e.g., 128 bits for a 128-bit AES key). Note that the RBG2(P) and RBG3 constructions only count entropy using Method 1 (see Sec. 5 and 6, respectively).

2.4. DRBGs

Approved DRBGs are specified in SP 800-90A. A DRBG includes instantiate, generate, and health-testing functions and may also 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 randomness into the DRBG's internal state and is used to recover from a potential (or actual) compromise (see Sec. 2.4.2 for an 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 are used to implement a DRBG design. The same implementation can be used to create multiple (logical) "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 (i.e., a pointer) that is unique to that instantiation (see Fig. 1). Each instantiation may be considered a different DRBG, even though it uses the same software code or hardware.

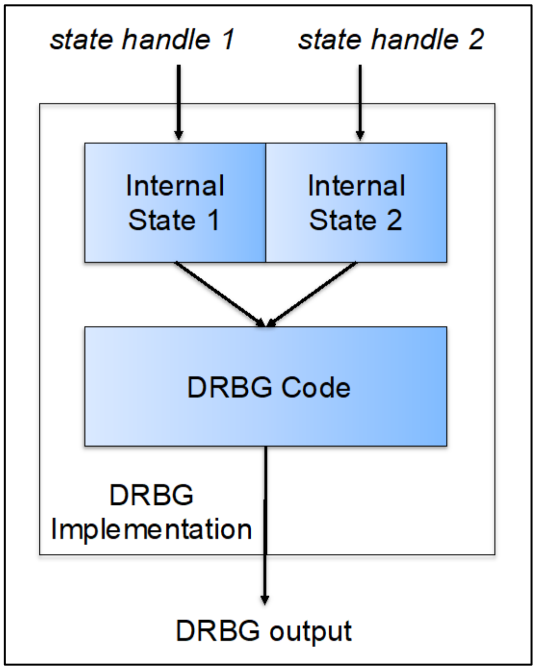

Fig. 1. DRBG instantiations

Each DRBG instantiation is initialized with input from some randomness source that establishes the security strength(s) 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 SP 800-90A.

A DRBG may be implemented to accept additional 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] A capability to handle additional input is recommended for an implementation.

2.4.2. Reseeding, Prediction Resistance, and Compromise Recovery

Under some circumstances, the internal state of an RBG (containing the RBG's secret information) could be leaked to an adversary. This might happen as the result of a side-channel attack or a serious compromise of the computer on which the DRBG runs and may not be detected by the DRBG or any consuming application.

[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.

In order to limit damage due to a compromised state, all DRBGs in SP 800-90A 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 the DRBGs in SP 800-90A, the RBGs specified in this document also inherit this property.

DRBGs may be reseeded at any time to allow for recovery from a potential compromise. An adversary who knows the internal state of the DRBG before the reseed but who does not learn the seed material used for the reseed knows nothing about its internal state after the reseed. Reseeding allows a DRBG to recover from a leak of its internal state.

In order to reseed a DRBG at a security of s bits, new seed material is provided to the DRBG from either an entropy source or an RBG. If the seed material is provided by an entropy source, it must contain at least s bits of min-entropy. If the seed material is provided by an RBG, the RBG must support at least a security strength of s bits, and the seed material must be at least s bits long. Seed material from an entropy source will always be unpredictable; seed material from an RBG will be unpredictable if that RBG has not been compromised.

A DRBG output is said to have *prediction resistance* when the DRBG is reseeded with at least s bits of min-entropy immediately before the output is generated by the DRBG. The entropy for this reseeding process needs to be provided by either an entropy source or an RBG3 construction for prediction resistance to be provided.

When a target DRBG is reseeded using another DRBG as a randomness source, the target DRBG is not guaranteed to have prediction resistance. If the source and target DRBGs are both compromised, then reseeding the target DRBG from the other DRBG will allow the adversary to know the target DRBG's internal state. However, it is often a good idea to reseed a target DRBG from a source DRBG. If the source DRBG was not compromised, then the target DRBG's state will be unknown to the adversary after the reseed.

The RBG3 construction always provides prediction resistance on its outputs, as every n-bit output has n bits of entropy. The RBG2 construction can provide prediction resistance on its outputs when reseeding is supported. The RBG1 construction never provides prediction resistance since it cannot be reseeded. Prediction resistance may be provided for the root RBGC construction but not for any subsequent non-root RBGC construction. However, subsequent RBGCs can (and generally **should**) periodically reseed from their randomness source (i.e., their parent).

The RBG1, RBG2, and RBGC constructions provide output with a security strength that depends on the security strength of the DRBG instantiation within the RBG and the length of the output. These constructions do not provide output with full entropy and **must not** be used by applications that require a higher security strength than has been instantiated in the DRBG of the construction. See Appendix A.1 for a discussion of entropy versus security strength.

Although reseeding provides fresh randomness that is incorporated into an already instantiated DRBG at a security strength of s bits, the reseed process does not increase 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.

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, 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 Sec. 2.8. The security boundary of a DRBG is discussed in SP 800-90A, and the security boundary for an entropy source is discussed in SP 800-90B. Both the entropy source and the DRBG contain their own health tests within their respective security boundaries.

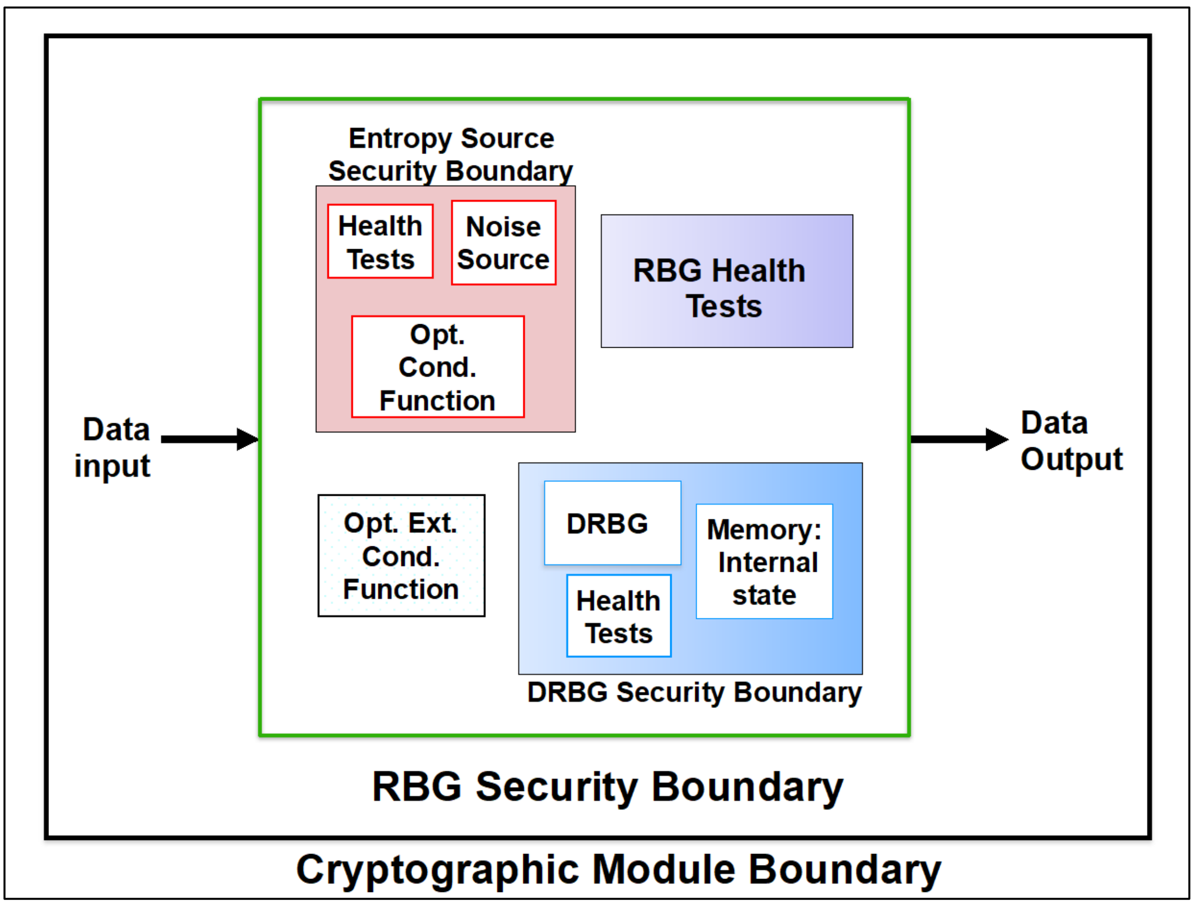

Fig. 2. Example of an RBG security boundary within a cryptographic module

Figure 2 shows an example RBG implemented within a FIPS 140-validated cryptographic module. In this figure, the RBG security boundary is completely contained within the cryptographic module boundary. The data input may be a personalization string or additional input (see Sec. 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 an (optional) external conditioning function. The RBG2 and RBG3 constructions in Sec. 5 and 6, respectively, use this model.

In the case of the RBG1 construction in Sec. 4, the security boundary containing the DRBG does not include a randomness source (shown as an entropy source in Fig. 2). For an RBGC construction, the security boundary is the computing platform on which the chain of DRBGs is used.

A cryptographic primitive (e.g., an **approved** hash function or block cipher) 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 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).
2. Entropy sources that have been validated for conformance to SP 800-90B are used to provide seed material for seeding and reseeding a DRBG or providing entropy for an RBG3 construction. The output of non-validated entropy sources is only used as additional input.

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

3. An entropy source outputs no more than 2^{64} bits. The number of output bits from the RBG is at most 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 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 SP 800-90B for entropy estimation.
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 SP 800-90B.

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 each independent entropy-source output that is contributing to the entropy in the bitstring (see Methods 1 and 2 in Sec. 2.3). For example, suppose that the outputs 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., $ES\_len_A + ES\_len_B + ES\_len_C$).
    - Using Method 1 in Sec. 2.3, the amount of entropy in the concatenated bitstring is $ES\_entropy_A + ES\_entropy_B$.
    - Using Method 2 in Sec. 2.3, the amount of entropy in the concatenated bitstring is the sum of all entropy 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 Sec. 3.2.2.2, 6.4, and 7 for the use of this assumption and IR 8427 for the rationale.
10. When entropy is requested, the entropy source responds as follows:
    - If the entropy source provides the requested amount of entropy, a *status* indication of success is returned along with a bitstring that contains the requested amount of entropy.
    - If the entropy source detects a failure of the primary noise source (i.e., an error from which it cannot recover), the entropy source returns a *status* indicating a failure. Other output is not provided.
    - If the entropy source indicates an error other than failure (e.g., entropy cannot be obtained in a timely manner, or there is an intermittent problem), the entropy source returns a *status* indicating that the entropy source cannot provide output at this time. Other output is not provided.

The following assumptions and assertions pertain to the use of DRBGs and the RBG constructions:

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 that are extracted by at least 64 bits. In particular, for a DRBG that has been instantiated at a security strength of s bits, s full-entropy bits can be extracted from the output of that DRBG when at least $s + 64$ bits of fresh entropy are inserted into the DRBG before the output is generated (see IR 8427).
12. To instantiate a DRBG at a security strength of s bits:

- For an RBG1 construction, a bitstring at least $3s/2$ bits long is needed from a randomness source (an RBG) providing at least s bits of security strength (see Sec. 4).
- For an RBG2 or RBG3 construction, bitstrings with at least $3s/2$ bits of entropy are needed from the entropy source(s) (see Sec. 5 and 6, respectively).
- For an RBGC construction that is the root of a tree of RBGC constructions, at least $3s/2$ bits of entropy are needed from the randomness source when the initial randomness source is a full-entropy source or RBG3 construction. If the initial randomness source is an RBG2 construction, a bitstring at least $3s/2$ bits long is needed from the randomness source (see Sec. 7).
- For an RBGC construction that is not the root of the tree, a bitstring at least $3s/2$ bits long is needed from the construction's randomness source (see Sec. 7).

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 Sec. 5 and 6) reside within the physical boundary of a single FIPS 140-validated cryptographic module.
15. All RBGC constructions in a DRBG chain reside on the same computing platform.
16. The DRBGs specified in SP 800-90A are assumed to meet their explicit security claims (e.g., backtracking resistance, claimed security strength, etc.).
17. A sub-DRBG is considered to be part of the RBG1 construction that initializes it.
18. The RBG1 construction and its sub-DRBGs reside within the physical boundary of a single FIPS 140-validated cryptographic module.

2.7. General Implementation and Use Requirements and Recommendations

When implementing the RBG constructions 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, and
3. **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 SP 800-90A, the entropy sources in SP 800-90B, and the RBG3 constructions specified in SP 800-90C are provided below and in Fig. 3.

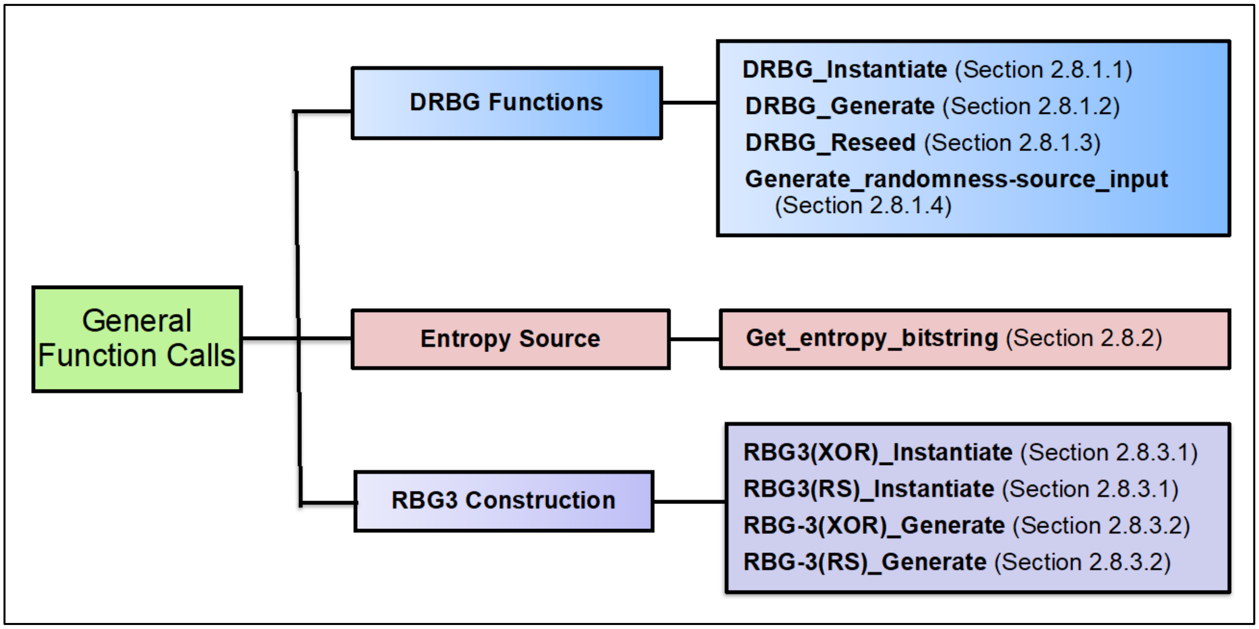

Fig. 3. General function calls

Each function returns a status code that **must** 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 item 10 in Sec. 2.6 and Sec. 8.1.2 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.

The distinction between a function within a DRBG or RBG and the request for the execution of that function by a requesting entity (e.g., an application) is needed for clarity. The requesting entity may not include an implementation of the function itself but needs to be able to request the DRBG or RBG to execute that function to obtain random values for its use. As used in this document, the request needs to provide some or all the input needed for the associated function. Relevant information output by that function needs to be returned in response to the request.

2.8.1. DRBG Functions

SP 800-90A specifies several functions within a DRBG that indicate the input and output parameters and other implementation details. In some cases, some input parameters identified in SP 800-90A may be omitted, and some output information may not be returned (e.g., because the requested information was not generated).

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 optional input (see Sec. 2.8.1.1) and
2. A generate function that produces output for use by a consuming application (see Sec. 2.8.1.2).

A DRBG may also support a reseed function (see Sec. 2.8.1.3).

A **Get_randomness-source_input** call is used in SP 800-90A to request output from a randomness source during instantiation and reseeding (see Sec. 2.8.1.4). The behavior of this function is specified in this document based on the type of randomness source used and the RBG construction.

The use of the **DRBG_Uninstantiate** function

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

(*status, state_handle*) = **DRBG_Instantiate** (*requested_instantiation_security_strength, personalization_string*).

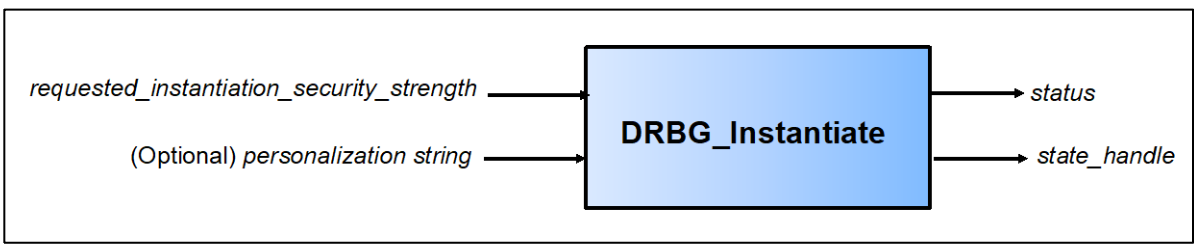

Fig. 4. DRBG_Instantiate function

The **DRBG_Instantiate** function (shown in Fig. 4) is used to instantiate a DRBG at the *requested_instantiation_security_strength* using the output of a randomness source[4] and an optional *personalization_string* to create a seed. As stated in Sec. 2.4.1, a *personalization_string* is optional but strongly recommended. Details about the **DRBG_Instantiate** function are provided in SP 800-90A.

If the *status* code returned for the **DRBG_Instantiate** function indicates a success (i.e., the DRBG has been instantiated at the requested security strength), a state handle may[5] be returned to

[4] The randomness source provides the seed material required to instantiate the security strength of the DRBG.

[5] 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.

indicate the particular DRBG instance (i.e., pointing to the internal state to be used by this instance). When provided by the **DRBG_Instantiate** function, the state handle is used in subsequent calls to the DRBG (e.g., during a **DRBG_Generate** call) to reference the internal state information for the instantiation. The information in the internal state includes the security strength of the instantiation and other information that changes during DRBG execution (see SP 800-90A for each DRBG design).

When the DRBG has been instantiated at the requested security strength, the DRBG will operate at that security strength even if the security strength requested in subsequent **DRBG_Generate** calls (see Sec. 2.8.1.2) is less than the instantiated security strength. For example, if a DRBG has been instantiated at a security strength of 256 bits, all output will be generated at that strength even when a request is received to generate bits at a strength of 128 bits.

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

The **DRBG_Instantiate** function is requested by an application using a **DRBG_Instantiate_request**:

(*status, state_handle*) = **DRBG_Instantiate_request**(*requested_instantiation_security_strength, personalization_string*).

As shown in Fig. 5, a **DRBG_Instantiate request** received by a DRBG results in the execution of the DRBG's instantiate function, providing the input parameters for that function. The **DRBG_Instantiate** function then obtains *seed_material* from the randomness source(s), instantiates a DRBG and returns the *status* of the process and (if there is no error) a *state_handle* for the internal state to the application.

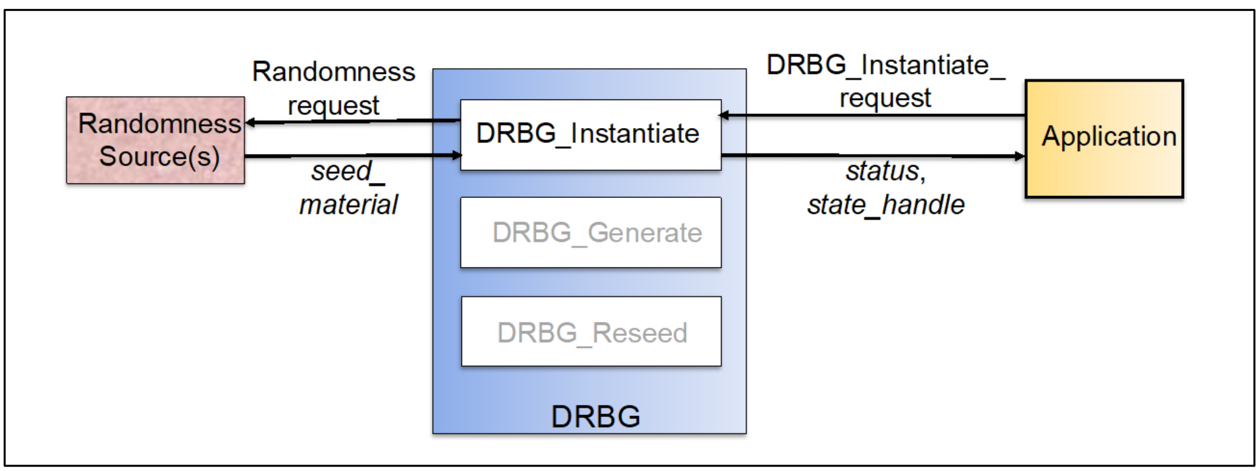

Fig. 5. DRBG_Instantiate request

2.8.1.1. DRBG Generation Request

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

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

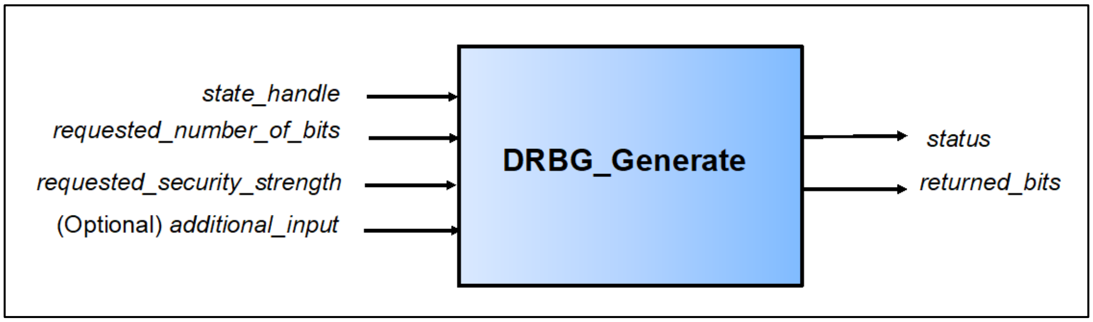

Fig. 6. DRBG_Generate function

The **DRBG_Generate** function (shown in Fig. 6) is used to generate a specified number of bits. If a suitable *state_handle* is available, it is included as input to indicate the DRBG instance to be used. The number of bits to be returned and the security strength that the DRBG needs to support for generating the bitstring are provided with (optional) additional input. As stated in Sec. 2.4.1, the ability to accept additional input is recommended.

The **DRBG_Generate** function returns status information — either an indication of success or an error. If the returned status code indicates a success, the requested bits are 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* consists of a *Null* bitstring. An example of a condition in which an error indication may be returned includes a request for a security strength that exceeds the instantiated security strength for the DRBG.

Details about the **DRBG_Generate** function are provided in SP 800-90A.

The **DRBG_Generate** function is requested by an application using a **DRBG_Generate_request**:

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

As shown in Fig. 7, a **DRBG_Generate_request** received by a DRBG results in the execution of the DRBG's **DRBG_Generate** function, providing the input parameters for that function. The **DRBG_Generate** function generates the requested number of bits and returns the *status* of the process and (if there is no error) the newly generated bits.

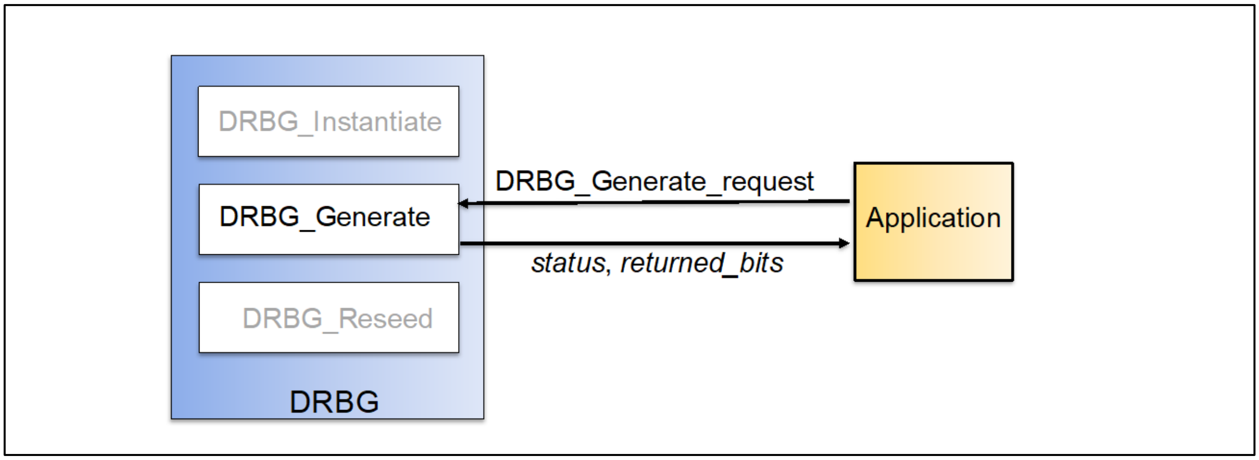

Fig. 7. DRBG_Generate_request

2.8.1.2. DRBG Reseed

The reseeding of a DRBG instantiation is intended to insert additional randomness into that DRBG instantiation (e.g., to recover from a possible compromise or to provide prediction resistance). This is accomplished using the following function:[6]

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

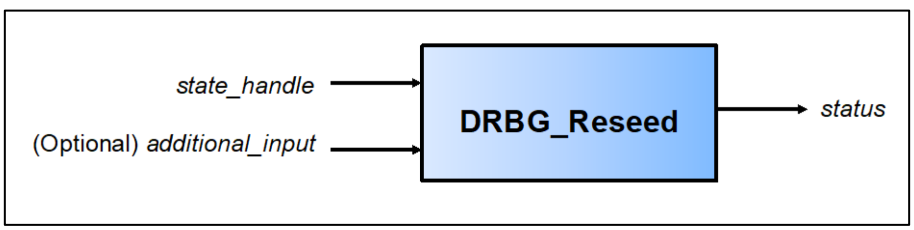

Fig. 8. DRBG_Reseed function

A **DRBG_Reseed** function (shown in Fig. 8) is used to acquire at least s bits of fresh randomness 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 to be reseeded.[7] In addition to the seed material provided from the DRBG's randomness source(s) during reseeding, optional *additional_input* may be incorporated into the reseed process. As discussed in Sec. 2.4.1, the capability for handling and using additional input is recommended. Details about the **DRBG_Reseed** function are provided in SP 800-90A.

[6] Note that this does not increase the security strength of the DRBG.
[7] The value of s may be available in the DRBG's internal state (see SP 800-90A).

An indication of the *status* is returned.

The **DRBG_Reseed** function is requested by an application using a **DRBG_Reseed_request**:

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

As shown in Fig. 9, a **DRBG_Reseed_request** received by a DRBG results in the execution of the DRBG's **DRBG_Reseed** function, providing the input parameters for that function. The **DRBG_Reseed** function then obtains *seed_material* from a randomness source, reseeds the DRBG instantiation, and returns the *status* of the process to the application.

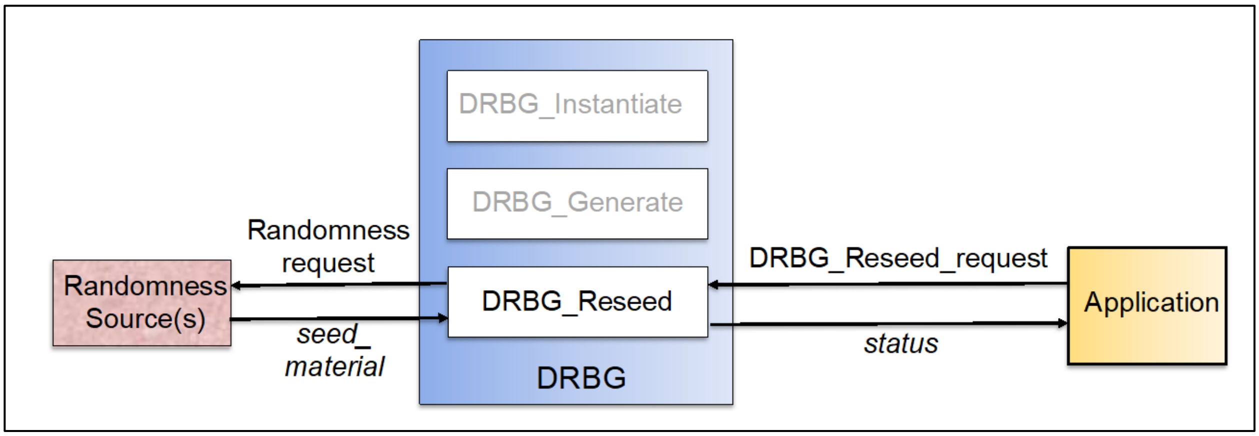

Fig. 9. DRBG_Reseed_request

2.8.1.3. Get_randomness-source_input Call

In SP 800-90A, a **Get_randomness-source_input** call is used in the **DRBG_Instantiate** function and **DRBG_Reseed** function to indicate when a randomness source needs to be accessed to obtain seed material. 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** be implemented based on various situations (e.g., the randomness source and the RBG construction used). Sections 3.2.2, 4, 5, 6, and 7 provide instructions for obtaining input from a randomness source when the **Get_randomness-source_input** call is encountered in SP 800-90A.

2.8.2. Interfacing With Entropy Sources

A single entropy source request 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 Sec. 6.4.1). SP 800-90C uses the term **Get_entropy_bitstring** to identify the process of obtaining the required entropy from one or more entropy sources. For convenience in describing the RBG constructions, this process is represented as a function whose input includes an indication of the amount of entropy that is needed from the entropy source(s) and whose output includes a status report on the success or failure of the process. If the process is successful, a bitstring containing the requested entropy is produced (see Fig. 10). The **Get_entropy_bitstring** function is invoked herein as:

(*status, entropy_bitstring*) = **Get_entropy_bitstring**(*bits_of_entropy, counting method, entropy_source_ID*),

where *bits_of_entropy* is the amount of entropy requested for return in the *entropy_ bitstring*, *counting_method* is the method to be used for counting entropy in the entropy source(s) (see Sec. 2.3), *entropy_source_ID* is an optional parameter that indicates the specific entropy source to be used, and *status* indicates whether the request has been satisfied.

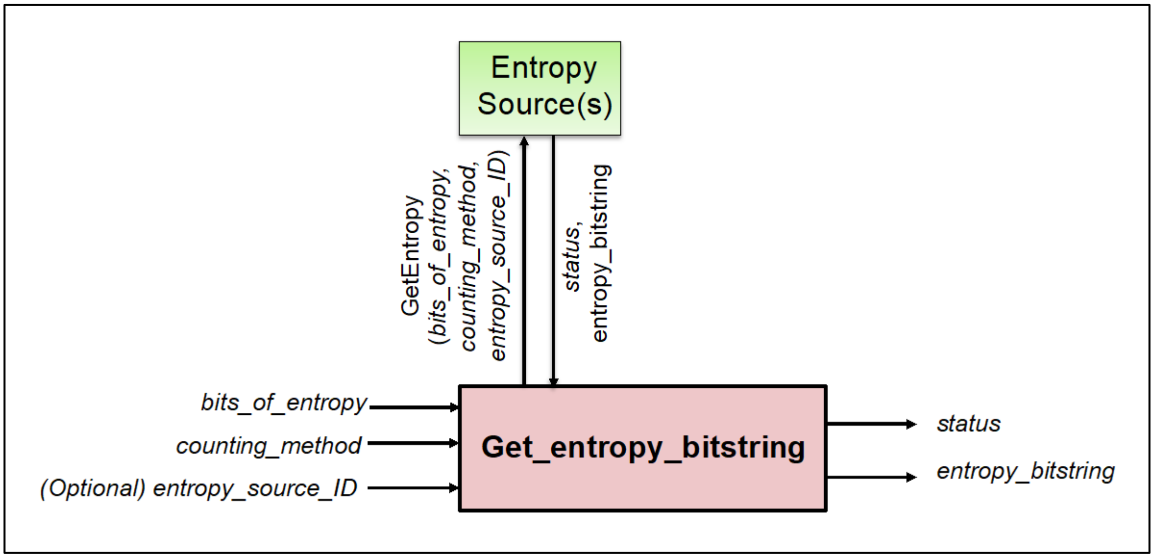

Fig. 10. Get_entropy_bitstring function

The **Get_entropy_bitstring** process requests entropy from whatever validated entropy sources are available or the entropy source identified by *entropy_source_ID* (if present). Any acquisition of entropy from non-validated entropy sources is handled separately (e.g., by a different process) to avoid misuse. See Sec. 3.1 for additional discussion about the **Get_entropy_bitstring** process.

2.8.3. Interfacing With an RBG3 Construction

An RBG3 construction requires functions to instantiate its DRBG (see Sec. 2.8.3.1) and to request the generation of full-entropy bits (see Sec. 2.8.3.2). The functions needed to access the DRBG itself are provided in Sec. 2.8.1.

2.8.3.1. Instantiating a DRBG Within an RBG3 Construction

The instantiate functions for the DRBG within the RBG3 constructions use the following functions:

(*status, state_handle*) = **RBG3(XOR)_Instantiate**(*requested_security_strength, personalization_string*)

and

(*status, state_handle*) = **RBG3(RS)_Instantiate**(*requested_security_strength, personalization_string*).

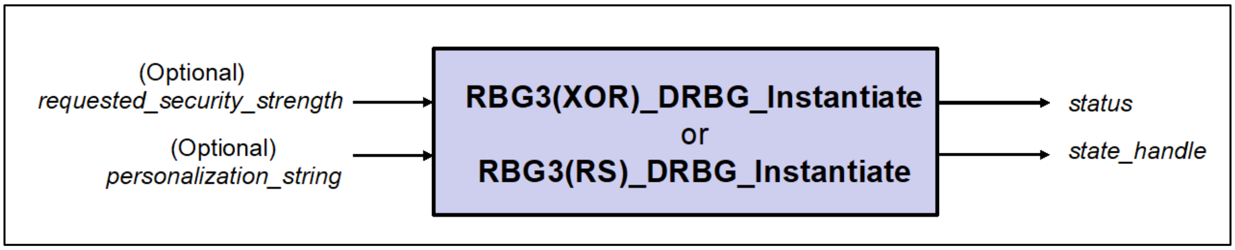

Fig. 11. RBG3 instantiate function

The instantiate function of the RBG3 construction (shown in Fig. 11) will result in the execution of the DRBG's instantiate function (provided in Sec. 2.8.1.1). A *requested_security_strength* may optionally be provided as an input parameter to indicate the minimum security strength to be supported by the DRBG within the RBG3 construction. An optional but recommended *personalization_string* (see Sec. 2.4.1) may be provided as an input parameter. If included as input to the RBG3 instantiation function, the *personalization_string* is passed to the DRBG that is instantiated by the instantiate function. See Sec. 6.4.1.1 and 6.5.1.1 for more specificity.

If the returned status code indicates a success, a state handle may be returned to indicate the DRBG instance that is to be used by the construction (i.e., the state handle points to the internal state used by this instance of the DRBG within the RBG3 construction). If multiple instances of the DRBG are used (in addition to the DRBG instance used by the RBG3 construction), a separate state handle is returned for each instance. When provided, the state handle is used in subsequent calls to that RBG (e.g., during a call to the RBG3 generate function; see Sec. 2.8.3.2) or when accessing the DRBG directly (e.g., during a reseed of the DRBG; see Sec. 6.4.1.4). 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 is returned.

The instantiation of the DRBG within an RBG3(XOR) or RBG3(RS) construction is requested by an application using an **Instantiate_RBG3_DRBG_request**:

(*status, state_handle*) = **Instantiate_RBG3_DRBG_request**(*requested_security_strength, personalization_string*).

Both the *requested_security_strength* and a *personalization_string* are optional in the **Instantiate_RBG3_DRBG_request**. As shown in Fig. 12, an **Instantiate_RBG3_DRBG_request** received by an RBG3 construction results in the execution of the DRBG's instantiate function.

The security strength of the DRBG within an RBG3 construction is the highest security strength that can be supported by the DRBG design (see Sec. 6). The *requested_security_strength* parameter in the **Instantiate_RBG3_DRBG_request should** be interpreted (in the case of the RBG3 construction) as the minimum security strength that is required by the consuming application if entropy-source failures are undetected. Therefore, if the *requested_security_strength* parameter is provided as input, it is compared against the value of the highest security strength that can be supported by the DRBG. If the *requested_security_strength* exceeds the security strength that can be supported by the DRBG, then an error indication is returned as the *status* in response to the **Instantiate_RBG3_DRBG_request**.

If no error is detected in the request, the **Instantiate_RBG3_DRBG** function obtains *seed_material* from the entropy source(s), instantiates the DRBG, and returns the *status* of the process and (possibly) a *state_handle* for the internal state to the application.

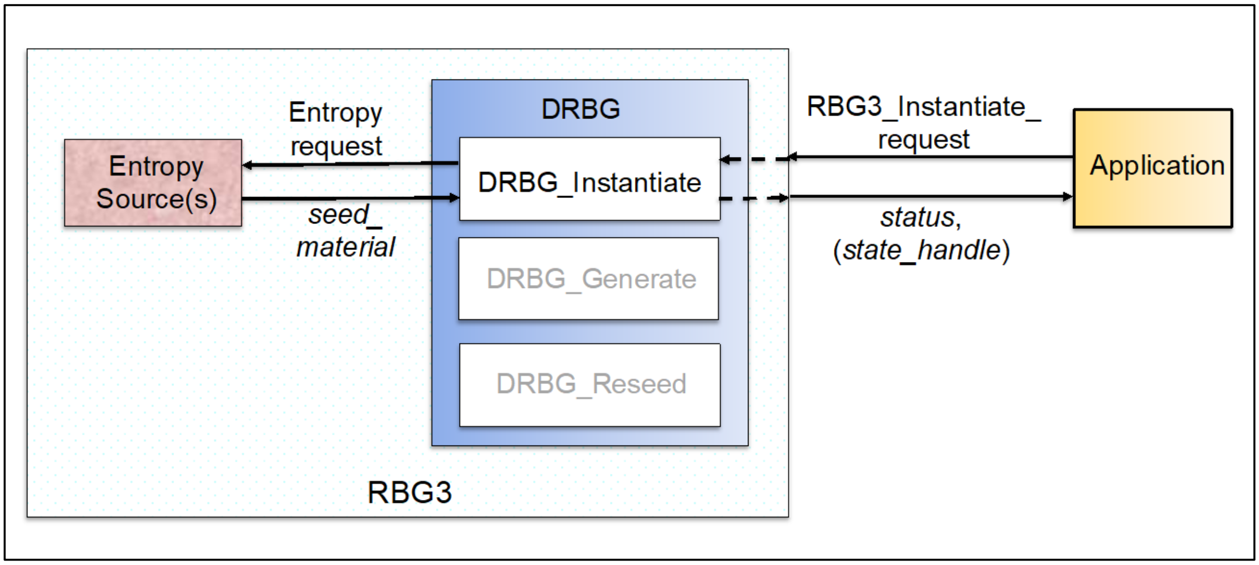

Fig. 12. RBG3(XOR) or RBG3(RS) instantiation request

2.8.3.2. Generation Using an RBG3 Construction

The RBG3(XOR) and RBG3(RS) generate function calls are essentially the same, but the function designs are very different (see Sec. 6.4 for the **RBG3(XOR)_Generate** function and Sec. 6.5 for the **RBG3(RS)_Generate** function):

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

and

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

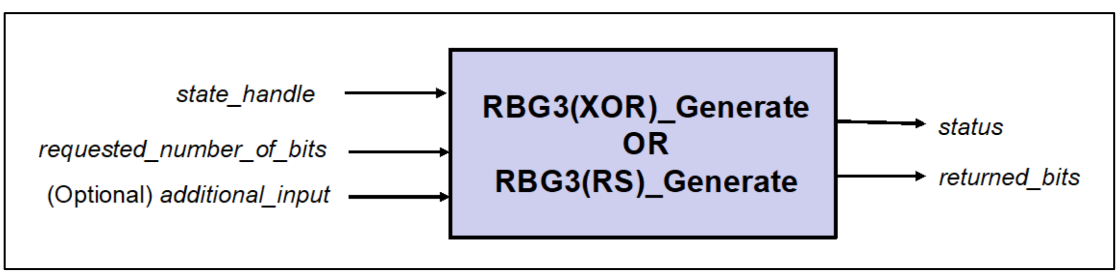

Fig. 13. RBG3 generate functions

The RBG3 generate functions are requested to use the DRBG indicated by the *state_handle* to generate the *requested_number_of_bits* using any (optional) *additional_input* provided. If the returned *status* code from the **RBG3(XOR)_Generate** or **RBG3(RS)_Generate** function indicates a success, a bitstring that contains the newly generated bits is also returned. If the status code indicates an error (e.g., the entropy source has failed), a *Null* bitstring is returned as the *returned_bits*.

The generation of random bits by an RBG3 construction is requested using the following:

(*status, returned_bits*) = **RBG3_Generate_ request**(*state_handle, requested_number_of_bits, requested_security_strength, additional_input*).

If a suitable *state_handle* is available (e.g., provided in response to an **Instantiate_RBG3_DRBG_request**; see Sec. 2.8.3.1), it is included in the **RBG3_Generate_request**. As shown in Fig. 14, an RBG3 generate request received by an RBG3 construction results in the execution of the RBG's generate function, providing the input parameters for that function. The entropy source is accessed, the requested number of bits are generated, and the *status* of the process and the newly generated bits are returned to the application. The RBG3 generate process for the RBG3(XOR) and RBG3(RS) construction are provided in Sec. 6.4 and 6.5, respectively.

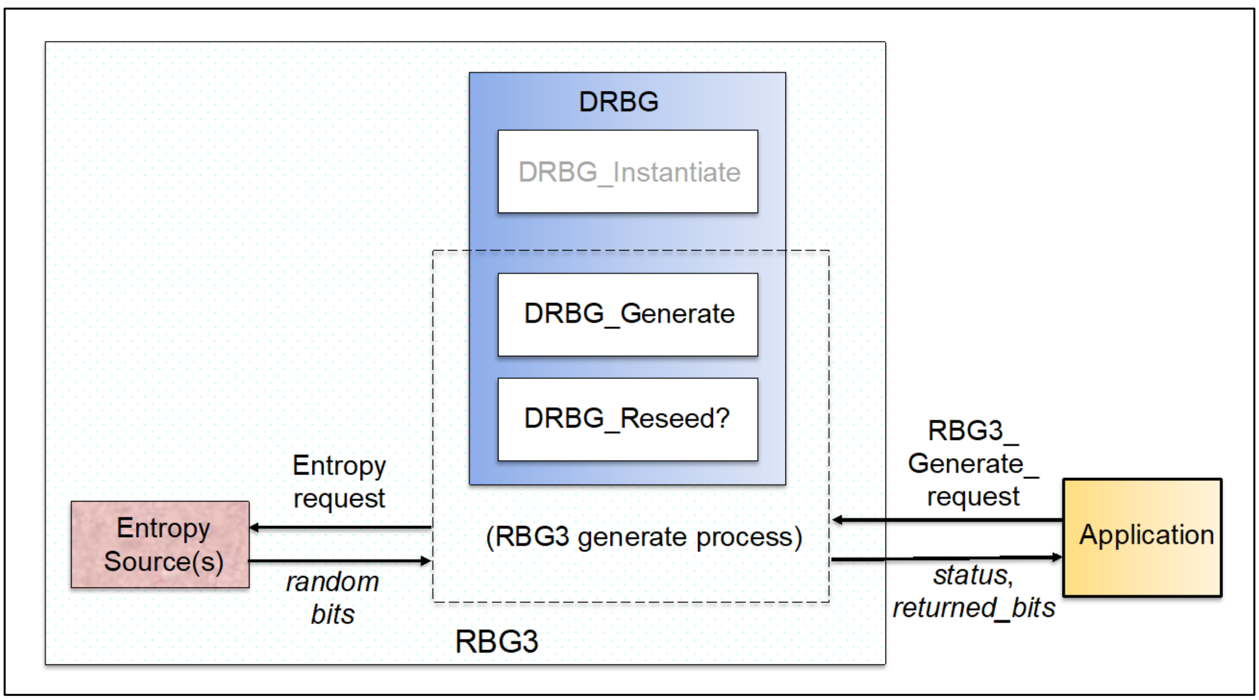

Fig. 14. Generic RBG3 generation process

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_entropy_bitstring** process to request entropy from one or more entropy sources. Section 3.2 discusses the conditioning of the output of one or more entropy sources before further use by an RBG.

3.1. Get_entropy_bitstring Process

The **Get_entropy_bitstring** process introduced in Sec. 2.8.2 obtains entropy from either 1) a designated entropy source or 2) one or more validated entropy sources in whatever manner is required (e.g., polling the entropy sources, waiting for an entropy source to provide output, or extracting bits that contain entropy from a pool of collected bits). The method for counting entropy from one or more entropy sources is indicated as an input parameter.

In many cases, the **Get_entropy_bitstring** process will need to query an entropy source (or a set of entropy sources) multiple times to obtain the amount of entropy requested. The details of the process are not specified in this document but are left to the developer to implement appropriately for the selected entropy source(s). However, the following behavior of the **Get_entropy_bitstring** process includes the following:

1. The **Get_entropy_bitstring** process **shall** only be used to access one or more validated entropy sources. Non-validated entropy sources **shall** be accessed by a separate process to avoid possible misuse.
2. Each validated entropy source **shall** be independent of all other validated or non-validated entropy sources used by the RBG.
3. The output produced from multiple entropy-source calls to a single validated entropy source or by calls to multiple independent, validated entropy sources **shall** be concatenated into a single bitstring. The entropy in the bitstring is the sum of the entropy provided by the validated entropy sources that are to be credited for contributing entropy to the process. For Method 1 (see Sec. 2.3), 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.
4. If a failure is reported during the **Get_entropy_bitstring** process by any physical or non-physical entropy source whose entropy is counted toward fulfilling an entropy request, the **Get_entropy_bitstring** process **shall** behave as follows (note that a bitstring containing entropy **should not** have been provided by that entropy source when a failure was reported; see Sec. 2.6, item 10):
    a. Method 1 is used for counting the entropy from one or more physical entropy sources:
        1) If a physical entropy source reports a failure, the error **shall** be reported to the consuming application as soon as possible. Any entropy collected

during the execution of the **Get_entropy_bitstring** process in which the error is reported **shall not** be used. This failed entropy source **shall not** be accessed to obtain entropy until the condition that caused the failure has been corrected and operational tests have been successfully passed.

If multiple physical entropy sources are used, the report **shall** identify the entropy source that reported the failure.

2) If a non-physical entropy source reports a failure, the failure may be ignored or reported to the consuming application along with a notification of the entropy source that failed. RBG operation may continue.

3) If all physical entropy sources report failures, RBG operation **shall** be terminated (i.e., stopped). The RBG **must not** be returned to normal operation until the conditions that caused the failures have been corrected and operational tests have been successfully passed.

4) If any physical entropy source is still "healthy" (i.e., the entropy source has not reported a failure), the RBG operations may continue using any healthy physical entropy source.

b. Method 2 in Sec. 2.3 is used for counting the entropy from one or more non-physical and/or physical entropy sources:

1) A failure from any entropy source **shall** be reported to the consuming application. If multiple entropy sources are used, the report **shall** identify the entropy source that reported the failure. This failed entropy source **shall not** be accessed to obtain entropy until the condition that caused the failure has been corrected and operational tests have been successfully passed.

2) If all entropy sources have reported failures, the RBG operation **shall** be terminated. The RBG **must not** be returned to normal operation until the conditions that caused the failures have been corrected and operational tests have been successfully passed.

3) If any physical or non-physical entropy source is still "healthy" (i.e., the entropy source has not reported a failure), RBG operations may continue using any healthy entropy source.

5. The **Get_entropy_bitstring** process **shall not** provide output for RBG operations unless the bitstring contains sufficient entropy to fulfill the entropy request.

3.2. External Conditioning

Entropy bits produced by one or more entropy sources are required for seeding and reseeding the DRBG in the RBG constructions specified in this document. Whether or not entropy-source output was conditioned within a validated entropy source prior to output, the entropy provided by the validated entropy source(s) may need to be conditioned prior to subsequent use by the RBG. For example:

- The entropy source within an RBG2 or RBG3 construction (see Sec. 5 or 6, respectively) is used to seed and reseed its DRBG. The entropy source may, for example, produce bitstrings that are too long for the specific DRBG implementation.
- Seed material with full entropy is required when the CTR_DRBG is implemented without a derivation function and an entropy source is used for seeding and reseeding the DRBG. If the entropy sources does not provide full-entropy output, the output needs to be conditioned prior to subsequent use by the DRBG to obtain full-entropy input for the DRBG.
- When the root RBGC construction in a DRBG chain uses a full-entropy source as its initial randomness source (see Sec. 7), the output from the entropy source(s) may need to be conditioned to provide a full-entropy bitstring for seeding and reseeding the root (i.e., the entropy source itself may not provide full-entropy output).
- If both physical and non-physical entropy sources are used to provide seed material, the entropy within the concatenated bitstring produced by these sources may not be distributed uniformly throughout the bitstring.

Since this conditioning is performed outside an entropy source, the output is said to be *externally conditioned*.

The conditioning function operates on a bitstring that is produced by the **Get_entropy_bitstring** process to produce an *entropy_bitstring*. Reasons to perform conditioning might include:

- Reducing the bias in the *entropy_bitstring*,
- Distributing entropy uniformly across the *entropy_bitstring*,
- Reducing the length of the *entropy_bitstring* and compressing the entropy into a smaller bitstring, and/or
- Ensuring the availability of full-entropy bits.

When external conditioning is performed, a vetted conditioning function listed in SP 800-90B **shall** be used. Additional vetted conditioning functions may be approved in the future.

3.2.1. Conditioning Function Calls

The conditioning functions operate on bitstrings obtained using the **Get_entropy_bitstring** process (see Section 3.1) to obtain an *entropy_bitstring* from one or more entropy sources.

The following format is used in Section 3.2.2 for a conditioning-function call:

conditioned_output_block = **Conditioning_function**(*input_parameters*), where the *input_parameters* for the selected conditioning function are discussed in Sections 3.2.1.2 and 3.2.1.3, and *conditioned_output_block* is the output returned by the conditioning function.

3.2.1.1. 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*), depending on the conditioning function and its primitive. Unlike other cryptographic applications, keys used in these external conditioning functions do not require secrecy to accomplish their purpose, so they 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 (i.e., *output_len*).

Table 2. Key lengths for the hash-based conditioning functions

Hash Function	Length of the output (*output_len*) and key (*keylen*)
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 (e.g., by obtaining 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). Any entropy used to randomize the key **shall not** be used for any other purpose.

3.2.1.2. 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:

   $$conditioned\_output\_block = \mathbf{Hash}(entropy\_bitstring),$$

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

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

   $$conditioned\_output\_block = \mathbf{HMAC}(Key,\ entropy\_bitstring),$$

   where HMAC operates on the *entropy_bitstring* using a *Key* determined as specified in Sec. 3.2.1.1.

In both 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. Using **Hash_df**, as specified in SP 800-90A:

   $$conditioned\_output\_block = \textbf{Hash\_df}(entropy\_bitstring,\ output\_len),$$

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

3.2.1.3. Block Cipher-Based Conditioning Functions

Conditioning functions may be based on **approved** block ciphers.[8] TDEA **shall not** be used as the block cipher.

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

1. Using CMAC (as specified in SP 800-38B) with an **approved** block cipher:

   $$conditioned\_output\_block = \textbf{CMAC}(Key,\ entropy\_bitstring),$$

   where CMAC operates on the *entropy_bitstring* using a *Key* determined as specified in Sec. 3.2.1.1.

2. Using CBC-MAC (specified in SP 800-90B) with an **approved** block cipher:

   $$conditioned\_output\_block = \textbf{CBC-MAC}(Key,\ entropy\_bitstring),$$

   where CBC-MAC operates on the *entropy_bitstring* using a *Key* determined as specified in Sec. 3.2.1.1.

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

   1. 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.[9]
   2. If the CBC-MAC conditioning function is used for the external conditioning of an entropy source output 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.

3. Using the **Block_cipher_df** as specified in SP 800-90A with an **approved** block cipher:

[8] 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.

[9] Any padding required could be done before submitting the *entropy_bitstring* to the CBC-MAC function.

conditioned_output_block = **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).

3.2.2. Using a Vetted Conditioning Function

There are several cases in which the use of an external conditioning function is required to prepare the entropy-source output for use by a DRBG mechanism. Section 3.2.2.1 provides a procedure for obtaining entropy from one or more entropy sources and subsequently processing it using an external conditioning function when full-entropy output is not required from the conditioning function (e.g., the conditioning function is used to compress the entropy into a shorter bitstring or to distribute the entropy across the output). Section 3.2.2.2 provides a procedure for obtaining full entropy from the entropy source(s) when needed. When full entropy is not required, either procedure may be used.

3.2.2.1. External Conditioning When Full Entropy is Not Required

The **Get_conditioned_input** procedure specified below iteratively requests entropy from the **Get_entropy_bitstring** process (represented as a **Get_entropy_bitstring** procedure; see Sec. 2.8.2 and 3.1) and distributes the entropy in the newly acquired *entropy_bitstring* across the conditioning function's output block. The output of the **Get_conditioned_input** procedure is the concatenation of the conditioning function output blocks. The entire output of the **Get_conditioned_input** procedure **shall** be provided as input to the DRBG mechanism (i.e., the output of the **Get_conditioned_input** function **shall not** be truncated).

Let *output_len* be the length of the conditioning function's output block.

Get_conditioned_input:

Input:

1. *n*: The amount of entropy to be obtained.
2. *counting_method*: The counting method to be used (i.e., either Method 1 or Method 2, as described in Sec. 2.3).
3. *target_entropy_source*: An optional parameter that indicates the specific entropy source to be queried. If the *target_entropy_source* is not indicated, output is to be obtained from any validated entropy sources producing output that have not reported a failure.

Output:

1. *status*: The status returned from the **Get_conditioned_input** process.

2. *Conditioned_entropy_bitstring*: A bitstring containing conditioned entropy or the *Null* string.

Process:

1. $v = \lceil n/output\_len \rceil$.
2. $w = \lceil n/v \rceil$.
3. *Conditioned_entropy_bitstring* = the *Null* string.
4. For $i = 1, ..., v$
   - 4.1 (*status, entropy_bitstring*) = **Get_entropy_bitstring**(*w, counting_method, target_entropy_source*).
   - 4.2 If (*status* ≠ SUCCESS), then return (*status*, *Null*).
   - 4.3 *conditioned_output_block* = **Conditioning_function**(*input_parameters*).
   - 4.4 *Conditioned_entropy_bitstring* = *Conditioned_entropy_bitstring* || *conditioned_output_block*.
5. Return (SUCCESS, *Conditioned_entropy_bitstring*).

Step 1 determines the number of output blocks (*v*) required to hold the requested amount of entropy.

Step 2 determines the amount of entropy (*w*) that will be requested for each of the *v* output blocks.

Step 3 sets the bitstring into which conditioned output will be collected (i.e., *Conditioned_entropy_bitstring*) to the *Null* string.

Step 4 is iterated *v* times to obtain and condition the requested amount of entropy for each output block of the conditioning function.

- Step 4.1 requests *w* bits of entropy from the entropy source(s) using the **Get_entropy_bitstring** call (see Sec. 2.8.2 and 3.1), indicating the method to be used for counting entropy (i.e., Method 1 or Method 2) and (if provided as input) the entropy source to be used (indicated by the *target_entropy_source* input parameter).
- Step 4.2 checks whether the *status* returned in step 4.1 indicated a success. If the *status* did not indicate a success, the *status* is returned with a *Null* string as the *Conditioned_entropy_bitstring*.
- Step 4.3 invokes the conditioning function for processing the *entropy_bitstring* obtained from step 4.1 to distribute the entropy throughout the conditioning function's output block. The *input_parameters* for the selected **Conditioning_function** are specified in Sec. 3.2.1.2 and 3.2.1.3 based on the conditioning function used.
- Step 4.4 concatenates the *conditioned_output_block* from step 4.3 to the *Conditioned_entropy_bitstring*.

- If all the requested entropy has not been obtained and conditioned, then go to step 4.1 with an updated value of *v*.

Step 5 returns a *status* of SUCCESS and the value of *Conditioned_entropy_bitstring*.

3.2.2.2. Conditioning Function to Obtain Full-Entropy Bitstrings

The **Get_conditioned_full_entropy_input** procedure specified below produces a bitstring with full entropy using one of the conditioning functions identified in Sec. 3.2.1 whenever a bitstring with full entropy is required. This process is unnecessary if full-entropy output is provided by the the entropy source(s).

The approach used by this procedure is to acquire sufficient entropy from the entropy source(s) to iteratively 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 in Sec. 2.6). This process is repeated until the requested number of full-entropy bits has been produced.

The **Get_conditioned_full_entropy_input** procedure obtains entropy from either 1) a designated entropy source (if a specific entropy source is identified as the *target_entropy_source*) or 2) any available entropy source using the **Get_entropy_bitstring** process (represented as a **Get_entropy_bitstring** procedure; see Sec. 2.8.2 and 3.1) and conditions the newly acquired *entropy_bitstring* to provide an *n*-bit string with full entropy.

Get_conditioned_full_entropy_input:

Input:

1. *n*: The amount of entropy to be obtained.
2. *counting_method*: The counting method to be used (i.e., either Method 1 or Method 2, as described in Sec. 2.3).
3. *target_entropy_source*: An optional parameter that indicates the specific entropy source to be queried. If the *target_entropy_source* is not indicated, output is to be obtained from any validated entropy sources producing output that have not reported a failure.

Output:

1. *status*: The status returned from the **Get_conditioned_full_entropy_input** process.
2. *Full_entropy_bitstring*: An *n*-bit string with full entropy or the *Null* string.

Process:

1. *temp* = the *Null* string.
2. *ctr* = 0.
3. While *ctr* < *n*, do

3.1 (*status, entropy_bitstring*) = **Get_entropy_bitstring**(*output_len* + 64, *counting_method, target_entropy_source*).

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

3.3 *conditioned_output_block* = **Conditioning_function**(*input_parameters*).

3.4 *temp* = *temp* || *conditioned_output_block*.

3.5 *ctr* = *ctr* + *output_len*.

4. *Full_entropy_bitstring* = **leftmost**(*temp, n*).

5. Return (SUCCESS, *Full_entropy_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.

Step 3 obtains and processes the entropy for each iteration.

- Step 3.1 requests *output_len* + 64 bits of entropy from the validated entropy source(s) using the indicated method for counting entropy (i.e., Method 1 or Method 2) and (if present) using only the entropy source identified as the *target_entropy_source*. If the entropy source to be used is not identified, the entropy is to be obtained from all available entropy sources that have not reported a failure.
- Step 3.2 checks whether the *status* returned in step 3.1 indicated a success. If the *status* did not indicate a success, the *status* is returned along with a *Null* bitstring as the *Full_entropy_bitstring*.
- 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 Sec. 3.2.1.2 or 3.2.1.3, depending on the conditioning function used.
- Step 3.4 concatenates the *conditioned_output_block* received in step 3.3 to the temporary bitstring (*temp*).
- Step 3.5 increments the counter for the number of full-entropy bits that have been produced so far.
- If less than *n* full-entropy bits have 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 *Full_entropy_bitstring*.

4. RBG1 Construction 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 its DRBG being instantiated securely from an RBG with access to a physical entropy source that resides outside of the device.

The DRBG in an RBG1 construction is instantiated (i.e., seeded) only once using either an RBG2(P) construction (see Sec. 5) or an RBG3 construction (see Sec. 6). Since a randomness source is not available after DRBG instantiation, the DRBG within an RBG1 construction cannot be reseeded (i.e., prediction resistance and recovery from a compromise cannot be provided).

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 the DRBG within an RBG1 construction cannot be reseeded, the use of the DRBG is limited to the DRBG's seedlife (see SP 800-90A).

Optionally, subordinate DRBGs (i.e., sub-DRBGs) may be used within the security boundary of an RBG1 construction (see Sec. 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 DRBG in the RBG1 construction is the source of the randomness that is used to 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, so reseeding of the DRBG within a sub-DRBG is not provided. A sub-DRBG may, in fact, be a different instantiation of the DRBG design implemented within the RBG1 construction (see Sec. 2.4.1).

4.1. RBG1 Description

As shown in Fig. 15, 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. For convenience and clarity, the DRBG within the RBG1 construction will sometimes be referred to as $DRBG_1$. Note that the required health tests are not shown in the figure.

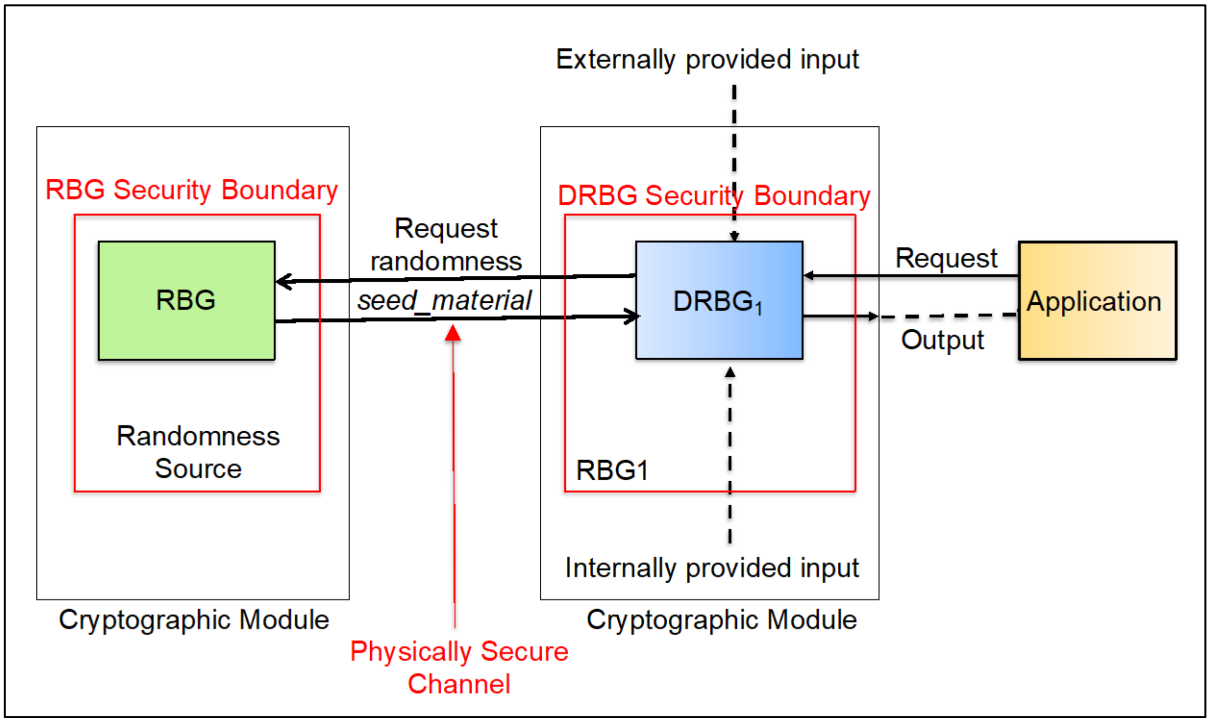

Fig. 15. Generic structure of the RBG1 construction

The RBG for instantiating $DRBG_1$ **must** be either an RBG2(P) construction that supports a reseed request from the RBG1 construction (see Sec. 5) or an RBG3 construction (see Sec. 6). A physically secure channel between the randomness source and $DRBG_1$ is used to securely transport the seed material required for DRBG instantiation. 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 Sec. 2.4.1).

An external conditioning function is not needed for this design because the output of the RBG used as the randomness source 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 Sec. 4.3) or by an application outside of the RBG1 security boundary. The security strength of the output produced by the RBG1 construction is the minimum of the security strengths provided by the DRBG within the construction and the RBG used as the randomness source 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 requests for instantiating the DRBG and generating pseudorandom bits (see Sec. 4.2.1 and 4.2.2, respectively). A reseed of the RBG1 construction cannot be performed because the randomness source is not available after instantiation.

4.2.1. Instantiating the DRBG in the RBG1 Construction

The DRBG within the RBG1 construction ($DRBG_1$) may be instantiated by an application at any security strength possible for the DRBG design using the **DRBG_Instantiate_request** discussed in Sec. 2.8.1.1:

(*status, RBG1_DRBG1_state_handle*) =
DRBG_Instantiate_request (*s, personalization_string*).

The **DRBG_Instantiate_request** received by $DRBG_1$ from an application **shall** result in the execution of the **DRBG_Instantiate** function within $DRBG_1$ (see Sec. 2.8.1.1):

(*status, RBG1_DRBG1_state_handle*) =
DRBG_Instantiate(*s, personalization_string*).

The *status* returned by the **DRBG_Instantiate** function **shall** be returned to the requesting application in response to the **DRBG_Instantiate_request**. *RBG1_ DRBG1_state_handle* is the state handle for $DRBG_1$'s internal state; the state handle may be *Null*.

The **DRBG_Instantiate** function within $DRBG_1$ **shall** use an external RBG (i.e., the randomness source) to obtain the *seed_material* necessary for establishing the DRBG's security strength.

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

If the randomness source is an RBG2(P) construction (see Fig. 16), the RBG2(P) construction **must** be reseeded using its internal entropy source(s) before generating bits to be provided to $DRBG_1$. The **Get_randomness-source_input** call in the **DRBG_Instantiate** function of $DRBG_1$ **shall** be replaced by a reseed request followed by a generate request to the RBG2(P) construction serving as the randomness source (see steps 1a and 2a below).

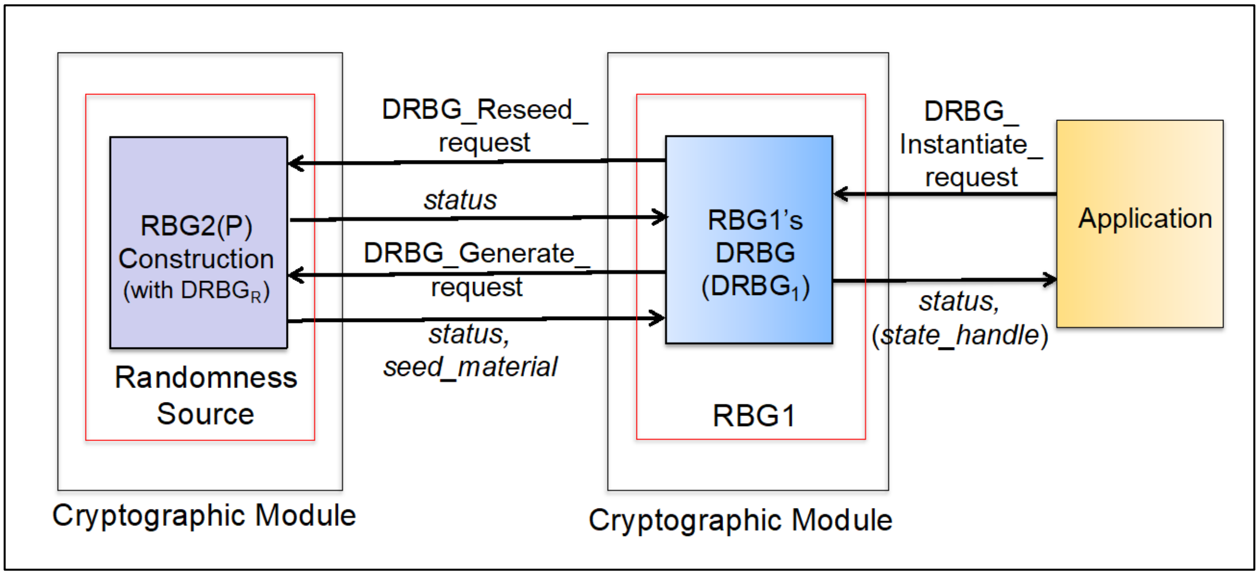

Fig. 16. Instantiation using an RBG2(P) construction as a randomness source

If the randomness source is an RBG3 construction (as shown in Fig. 17), the **Get_randomness-source_input** call in the **DRBG_Instantiate** function of $DRBG_1$ **shall** be replaced by the appropriate call to the RBG3 generate function (see Sec. 2.8.3.2, 6.4.1.2, and 6.5.1.2 and steps 1b and 2b below).

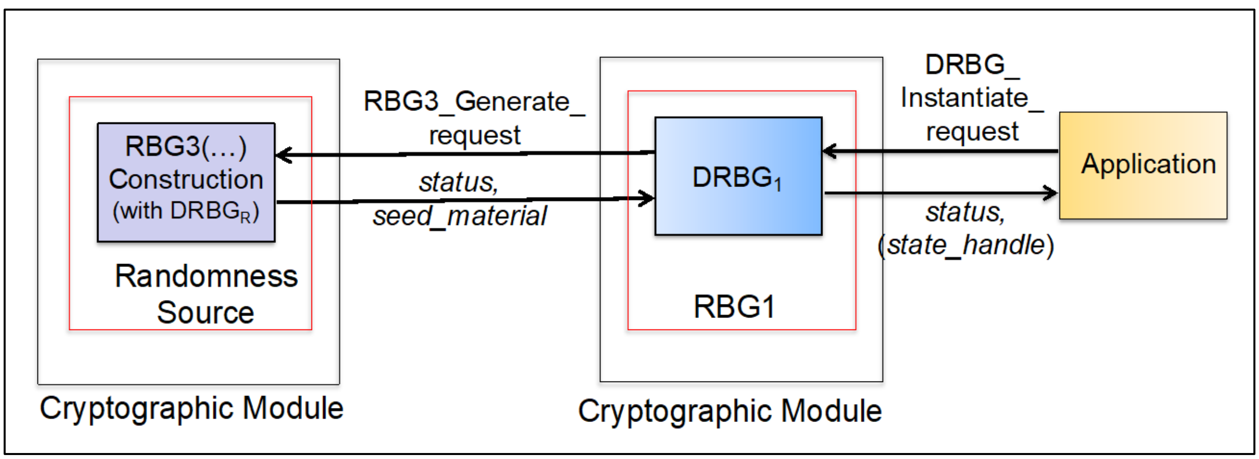

Fig. 17. Instantiation using an RBG3(XOR) or RBG3(RS) construction as a randomness source

Let $DRBG_1$ be the DRBG to be instantiated within the RBG1 construction and let $DRBG_R$ be the DRBG used within the randomness source (i.e., an RBG2(P) or RBG3 construction). Let s be the security strength to be instantiated for $DRBG_1$. **DRBG_Reseed_request** and **DRBG_Generate_request** are used below by an application to request the generation and reseed of the DRBG within the randomness source (i.e., $DRBG_R$). Let *DRBGR_state_handle* be the state handle for $DRBG_R$.

Upon receiving the instantiation request from the application, $DRBG_1$ is instantiated as follows:

1. When an RBG1 construction is instantiating a CTR_DRBG without a derivation function, $s + 128$ bits[10] **shall** be obtained from the randomness source as follows:

    a. If the randomness source is an RBG2(P) construction (see Fig. 16), the **Get_randomness-source_input** call in the **DRBG_Instantiate** function of $DRBG_1$ is replaced by a request to reseed $DRBG_R$ (the DRBG within the RBG2(P) construction), followed by a request to generate bits:

    - *status* = **DRBG_Reseed_request**(*$DRBG_R$_state_handle, additional_input*).
    - If (*status* ≠ SUCCESS), then return (*status, Invalid_state_handle*).
    - (*status, seed_material*) = **DRBG_Generate_request**(*$DRBG_R$_state_handle, s* + 128, *s, additional_input*).
    - If (*status* ≠ SUCCESS), then return (*status, Invalid_state_handle*).

    DRBG_Reseed_request and **DRBG_Generate_request** are used here to indicate requests for the DRBG within the randomness source ($DRBG_R$) to execute the **DRBG_Reseed** function and **DRBG_Generate** function within $DRBG_R$ (see Sec. 2.8.1.3, and 2.8.1.2, respectively). Also, see Sec. 5.2.3 and 5.2.2 for the handling of the reseed and generate requests by the RBG2(P) construction.

    b. If the randomness source is an RBG3(XOR) or RBG3(RS) construction (see Fig. 17), the **Get_randomness-source_input** call in the **DRBG_Instantiate** function of $DRBG_1$ is replaced by a request for the generation of random bits:

    - (*status, seed_material*) = **RBG3_Generate_ request**(*$DRBG_R$_state_handle, s* + 128, *additional_input*).
    - If (*status* ≠ SUCCESS), then return (*status, Invalid_state_handle*).

    RBG3_Generate_request is intended to result in the execution of the **DRBG_Generate** function in $DRBG_R$ (see Sec. 2.8.3.1). Also, see Sec. 6.4.1.2 and 6.5.1.2.1 for the handling of the generate request by the RBG3(XOR) and RBG3(RS) constructions, respectively.

2. When an RBG1 construction is instantiating any other DRBG (including a CTR_DRBG with a derivation function[11]), $3s/2$ 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 Fig. 16), the **Get_randomness-source_input** call in $DRBG_1$ is replaced by a request to reseed $DRBG_R$, followed by a request to generate bits:

---

[10] For AES, the block length is 128 bits, and the key length is equal to the security strength s. SP 800-90Ar1 requires the seed material from the randomness source to be key length + block length bits when a derivation function is not used.

[11] Although the use of a derivation function with the CTR_DRBG is allowed in an RBG1 construction, it is not needed to process output from the randomness source, since the randomness source is an RBG2(P) or RBG3 construction.

- *status* = **DRBG_Reseed_request**(*$DRBG_R$_state_handle, additional_input*).
- If (*status* ≠ SUCCESS), then return (*status, Invalid_state_handle*).
- (*status, seed_material*) = **DRBG_Generate_request**(*$DRBG_R$_state_handle,* 3*s*/2, *s, additional_input*).
- If (*status* ≠ SUCCESS), then return (*status, Invalid_state_handle*).

DRBG_Reseed_request and **DRBG_Generate_request** are used here to indicate requests for the DRBG within the randomness source ($DRBG_R$) to execute the **DRBG_Reseed** function and **DRBG_Generate** function within $DRBG_R$ (see Sec. 2.8.1.3 and 2.8.1.2, respectively). Also, see Sec. 5.2.3 and 5.2.2 for the handling of the reseed and generate requests by the RBG2(P) construction.

b. If the randomness source is an RBG3(XOR) or RBG3(RS) construction (see Fig. 17), the **Get_randomness-source_input** call in $DRBG_1$ is replaced by a request for the generation of random bits:

- (*status, seed_material*) = **RBG3_DRBG_Generate_request**(*$DRBG_R$_state_handle,* 3*s*/2, *additional_input*).
- If (*status* ≠ SUCCESS), then return (*status, Invalid_state_handle*).

RBG3_DRBG_Generate_request is intended to result in the execution of the **DRBG_Generate function** in $DRBG_R$ (see Sec. 2.8.3.1). Also, see Sec. 6.4.1.2 and 6.5.1.2.1 for the handling of the generate request by the RBG3(XOR) and RBG3(RS) constructions, respectively.

4.2.2. Requesting Pseudorandom Bits

As discussed in Sec. 2.8.1.2, an application requests the RBG1 construction to generate bits as follows:

(*status, returned_bits*) = **DRBG_Generate_request**(*RBG1_DRBG1_state_handle, requested_number_of_bits, s, additional_input*).

The **DRBG_Generate_request** results in the execution of the **DRBG_Generate** function within $DRBG_1$:

(*status, returned_bits*) = **DRBG_Generate**(*RBG1_DRBG1_state_handle, requested_number_of_bits, s, additional_input*).

The *status* returned by the **DRBG_Generate** function **shall** be returned to the requesting application. If the *status* indicates a successful process, the *returned_bits* **shall** also be provided to the application in response to the request.

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

Figure 18 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.

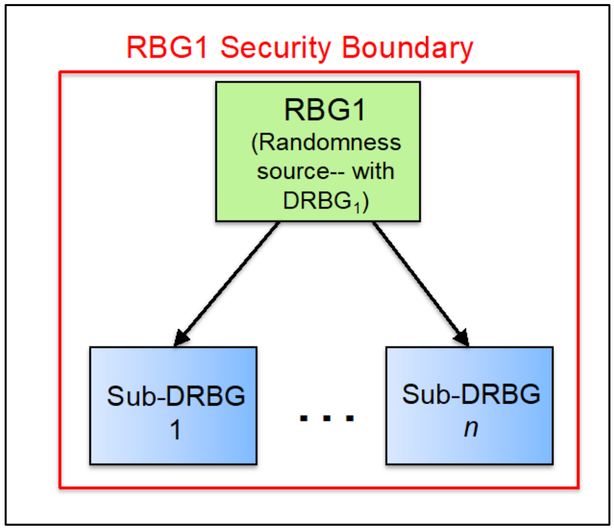

Fig. 18. RBG1 construction with sub-DRBGs

The RBG1 construction and each of its sub-DRBGs **shall** be implemented as separate physical or logical entities (see Fig. 18). Let $DRBG_1$ be the DRBG used by the RBG1 construction itself, with *RBG1_DRBG1_state_handle* used as the state handle for the internal state of $DRBG_1$. Let *sub-DRBGx_state_handle* be the state handle for the internal state of sub-DRBGx.

- When implemented as separate physical entities, the DRBG algorithms used by $DRBG_1$ and the sub-DRBGs **shall** be the same DRBG algorithm (e.g., the RBG1 construction and all its sub-DRBGs use HMAC_DRBG with 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.

The sub-DRBGs have the following characteristics:

1. Only one layer of sub-DRBGs is allowed.
2. Sub-DRBG outputs are considered outputs of the RBG1 construction.
3. The security strength that can be provided by a sub-DRBG is no more than the security strength of $DRBG_1$ (i.e., the DRBG within the RBG1 construction that is serving as the randomness source for the sub-DRBG).
4. Sub-DRBGs cannot provide output with full entropy.
5. The number of sub-DRBGs that can be instantiated by an RBG1 construction is limited only by the practical considerations associated with the implementation or application.

4.3.1. Instantiating a Sub-DRBG

An application may request the RBG1 construction to instantiate a sub-DRBG. The following represents the form of the application's request for sub-DRBG instantiation:

(*status, sub-DRBG_state_handle*) =
Instantiate_sub-DRBG_request(*s, personalization_string*).

$DRBG_1$ executes an **Instantiate_sub-DRBG** function. The *status* of the process is returned to the application with a state handle if the *status* indicates success.

The value of *max_personalization_string_length* is specified in SP 800-90A.

Instantiate_sub-DRBG:

Input:

1. *s*: the requested security strength for the sub-DRBG.
2. (Optional) *personalization_string*: An input that provides personalization information.

Output to a consuming application:

1. *status*: The status returned from the **Instantiate_sub-DRBG** function (see steps 2, 3, 6, and 10). If any status other than SUCCESS is returned, an invalid _state handle **shall** be returned.
2. *sub-DRBG_state_handle*: Used to identify the internal state for this sub-DRBG instantiation in subsequent calls to the generate function (see Sec. 4.3.2).

Information retained within the DRBG boundary after instantiation:

The internal states for $DRBG_1$ and the sub-DRBG instantiation.

Process:

1. Obtain the current internal state of $DRBG_1$ to get its instantiated security strength (shown as *RBG1_DRBG1_security_strength* in step 2).
2. If (*s* > *RBG1_DRBG1_security_strength*), then return (ERROR_FLAG**,** *Invalid_state_handle*).
3. If the length of the *personalization_string* > *max_personalization_string_length*, return (ERROR_FLAG, *Invalid_state_handle*).
4. If ($s > 192$), then $s = 256$

   Else, if ($s \le 128$), then $s = 128$.

   Else $s = 192$.

   Comment: See the instructions below for the value of *number_of_bits_to_generate*.

5. (*status, seed_material*) = **DRBG_Generate**(*RBG1_DRBG1_state_handle, number_of_bits_to_generate, s*).

6. If (*status* ≠ SUCCESS), return (*status*, *Invalid_state_handle*).

7. *working_state_values* = **Instantiate_algorithm**(*seed_material, personalization_string*).

8. Get the *sub-DRBG_state_handle* for a currently empty internal state. If an empty internal state cannot be found, return (ERROR_FLAG, *Invalid_state_handle*).

9. Set the internal state for the new instantiation (e.g., as indicated by *sub-DRBG_state_handle*):

   9.1 Record the *working_state_values* returned from step 7.

   9.2 Record any administrative information (e.g., the value of *s*).

10. Return (SUCCESS, *sub-DRBG_state_handle*).

Step 1 obtains DRBG$_1$'s security strength. A description of the internal state for each DRBG type is provided in SP 800-90A.

Steps 2 and 3 check the validity of the requested security strength *s* and the length of any personalization string provided for the instantiation request. An ERROR_FLAG and an invalid state handle are returned to the requesting application if either is unacceptable.

Step 4 sets the security strength to be established for the sub-DRBG instantiation based on the requested security strength *s*.

Step 5 requests the generation of *seed_material* at a security strength of *s* bits using DRBG$_1$. The *number_of_bits_to_generate* depends on DRBG$_1$'s type:

- When CTR_DRBG without a derivation function is implemented for DRBG$_1$, *number_of_bits_to_generate* = $s + 128$.
- Otherwise, *number_of_bits_to_generate* = $3s/2$.

Step 6 checks the *status* returned from step 5. If a *status* of SUCCESS is not returned, the *status* and an invalid state handle are returned to the requesting application.

Step 7 invokes the appropriate instantiate algorithm in SP 800-90A for DRBG$_1$'s design. Values for the working state portion of the sub-DRBG's internal state are returned by the instantiate algorithm.

Step 8 assigns a state handle for an available internal state. If no internal state is currently available, an ERROR_FLAG and invalid state handle are returned to the requesting application.

Step 9 enters the required values into the assigned internal state for the sub-DRBG.

Step 10 returns a *status* of SUCCESS and the assigned state handle to the requesting application.

4.3.2. Requesting Random Bits From a Sub-DRBG

As discussed in Sec. 2.8.1.2, pseudorandom bits may be requested from a sub-DRBG by an application:

(*status*, *returned_bits*) = **DRBG_Generate request**(*sub_DRBGx_state_handle*, *requested_number_of_bits*, *requested_security_strength*, *additional_input*).

The generate request received by the sub-DRBG **shall** result in the execution of the **DRBG_Generate** function:

(*status*, *returned_bits*) = **DRBG_Generate**(*sub_DRBGx_state_handle*, *requested_number_of_bits*, *requested_security_strength*, *additional_input*).

The *status* returned by the **DRBG_Generate** function **shall** be returned to the application in response to the request. If the process is successful, the newly generated bits (*returned_bits*) **shall** also be provided to the application in response to the **DRBG_Generate_request**.

4.4. Requirements

4.4.1. RBG1 Construction Requirements

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

1. An **approved** DRBG from SP 800-90A whose components can provide the targeted security strength for the RBG1 construction **shall** be employed.
2. The components of the RBG1 construction **shall** be successfully validated for compliance with SP 800-90A, SP 800-90C, FIPS 140, 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 capability to be reseeded.
5. The RBG1 construction **shall not** permit itself to be instantiated more than once.[12]
6. The randomness source **shall** be in a separate device from that of the RBG1 construction.
7. For CTR_DRBG (<u>with</u> a derivation function), Hash_DRBG, or HMAC_DRBG, $3s/2$ bits **shall** be obtained from a randomness source, where s is the targeted security strength for the DRBG used in the RBG1 construction ($DRBG_1$).

[12] While it is 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.

8. For a CTR_DRBG <u>without</u> a derivation function within the RBG1 construction, $s + 128$ bits[13] **shall** be obtained from the randomness source, where s is the targeted security strength for the DRBG used in the RBG1 construction ($DRBG_1$).

9. An implementation of an RBG1 construction **shall** verify that the internal state has been updated before the generated output is provided to the requesting entity.

10. The RBG1 construction **shall not** provide output for generating requests that specify a security strength greater than the instantiated security strength of its DRBG.

11. If the RBG1 construction can be used to instantiate a sub-DRBG, the RBG1 construction **may** directly produce output for an application in addition to instantiating a sub-DRBG.

12. Seed material produced by the RBG1 construction to instantiate a sub-DRBG **shall not** be used to instantiate other sub-DRBGs nor be provided directly to a consuming application.

13. If the seedlife of the DRBG within the RBG1 construction ($DRBG_1$) is ever exceeded or a health test of the DRBG fails, the use of the RBG1 construction **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.

14. A validated RBG2(P) construction with support for reseeding requests or a validated RBG3 construction **must** be used as the randomness source for the DRBG in the RBG1 construction ($DRBG_1$).

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 DRBG within the RBG1 construction ($DRBG_1$).

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. If an RBG2(P) construction is used as the randomness source for the RBG1 construction, the RBG2(P) construction **must** be reseeded before generating bits for each RBG1 instantiation.

18. A physically secure channel **must** be used to insert the seed material from the randomness source into the DRBG of the RBG1 construction ($DRBG_1$).

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, and a sub-DRBG **shall not** serve as a randomness source for another sub-DRBG.

---

[13] Note that $s + 128 = keylen + blocklen = seedlen$, as specified in SP 800-90Ar1.

2. A sub-DRBG **shall** employ the same DRBG components as its randomness source (i.e., the RBG1 construction).

3. A sub-DRBG **shall** reside in the same security boundary as the RBG1 construction that instantiates it.

4. The output from the RBG1 construction that is used for sub-DRBG instantiation **shall not** be output from the security boundary that contains the RBG1 construction and sub-DRBG and **shall not** be used for any other purpose, including for seeding a different sub-DRBG.

5. The security strength for a target sub-DRBG **shall not** exceed the security strength that is supported by the RBG1 construction.

6. For CTR_DRBG (with a derivation function), Hash_DRBG, or HMAC_DRBG, $3s/2$ bits **shall** be obtained from the RBG1 construction for instantiation of the sub-DRBG, where s is the requested security strength for the target sub-DRBG.

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

8. A sub-DRBG **shall not** produce output until it is instantiated.

9. 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.

10. An implementation of a sub-DRBG **shall** verify that the internal state has been updated before the generated output is provided to the requesting entity.

11. The sub-DRBG **shall not** be reseeded.

12. 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 (i.e., not testable by the validation labs) is:

13. The output of a sub-DRBG **must not** be used as seed material for other DRBGs (e.g., the DRBGs in other RBGs) or sub-DRBGs.

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 and generates outputs on demand. An RBG2 construction may (optionally) be implemented to support reseeding requests from a consuming application (i.e., providing prediction resistance for the next output of the RBG2 construction to mitigate a possible compromise of previous internal states) and/or to be reseeded in accordance with implementation-selected criteria.

If a consuming application requires full-entropy output, an RBG3 construction from Sec. 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 Fig. 19). One or more entropy sources are used to provide the entropy bits for both DRBG instantiation and any reseeding of the DRBG. The use of a personalization string and additional input is optional and may be provided from within the cryptographic module or from outside of that module.

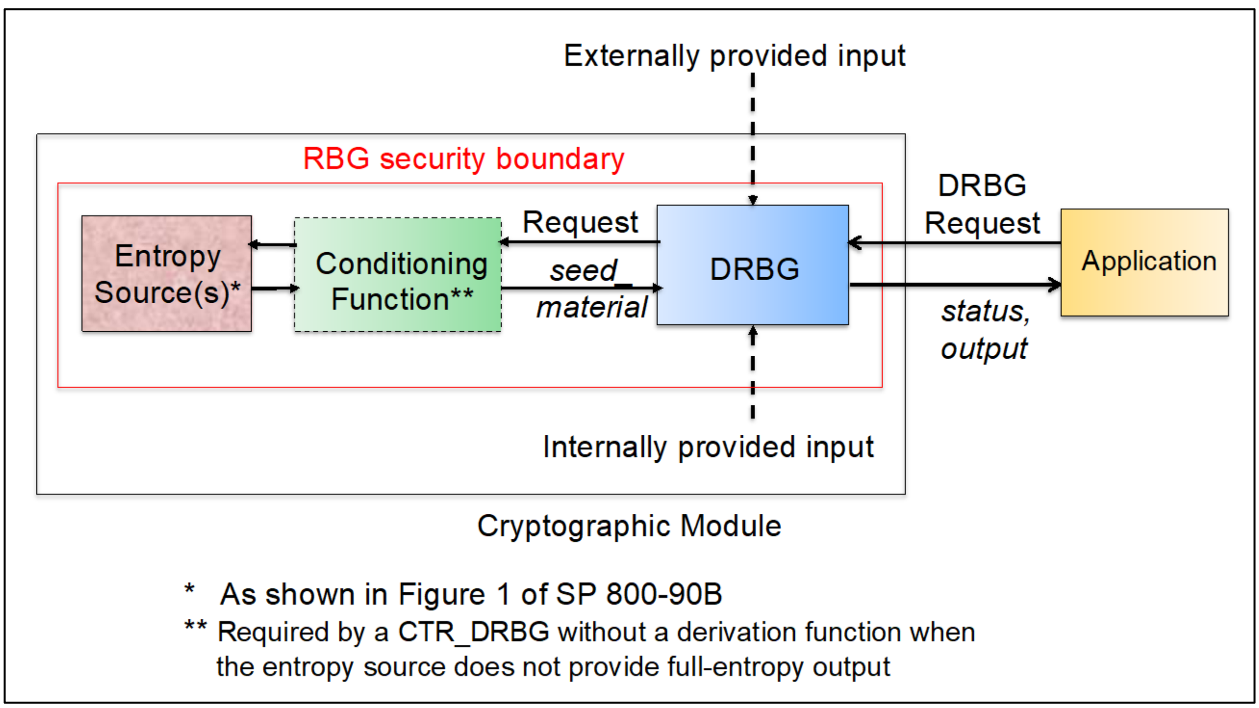

Fig. 19. Generic structure of the RBG2 construction

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 Sec. 2.3 (i.e., only the entropy produced by one or more 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. An RBG2(P) construction may exist as part of an RBG3 construction (see Sec. 6).
2. An RBG2(NP) construction uses the output of any validated non-physical or physical entropy source(s), as discussed in Method 2 of Sec. 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 may affect the implementation of a **Get_entropy_bitstring** process (represented as a **Get_entropy_bitstring** procedure; see Sec. 2.8.2 and 3.1), either accessing the entropy source(s) directly or via the **Get_conditioned_input** or **Get_conditioned_full_entropy_input** procedure specified in Sec. 3.2.2 during instantiation and reseeding (see Sec. 5.2.1 and 5.2.3). That is, when seeding and reseeding an RBG2(P) construction (including a DRBG within an RBG3 construction, as discussed in Sec. 6), Method 1 in Sec. 2.3 is used to combine the entropy from the entropy source(s), and Method 2 is used when instantiating and reseeding an RBG2(NP) construction.

5.2. Conceptual Interfaces

The RBG2 construction includes requests for instantiating the DRBG (see Sec. 5.2.1) and generating pseudorandom bits (see Sec. 5.2.2). Once instantiated, an RBG2 construction may be reseeded when requested by a consuming application or when determined by implementation-selected criteria if a reseed capability has been implemented (see Sec. 5.2.3).

5.2.1. RBG2 Instantiation

An RBG2 construction may be instantiated by an application at any valid[14] security strength possible for the DRBG design and its components using an instantiation request (see Sec. 2.8.1.1):

(*status, RBG2_DRBG_state_handle*) = **DRBG_Instantiate_request**(*s, personalization_string*).

The request results in the execution of the **DRBG_Instantiate** function within the DRBG:

[14] The security strength must be 128, 192, or 256 bits.

(*status, RBG2_DRBG_state_handle*) = **DRBG_Instantiate**(*s, personalization_string*).

The **DRBG_Instantiation** function returns the *status* of the process, which is then provided to the application in response to the request. If the process is successful, a state handle for the instantiation (e.g., *RBG2_DRBG_state_handle*) is also returned from the **DRBG_Instantiate** function and may be forwarded to the application.[15]

An RBG2 construction obtains entropy for its DRBG from one or more validated entropy sources within its boundary, either directly or using a conditioning function to obtain and process the output of the entropy source(s).

SP 800-90A uses a **Get_randomness-source_input** call in the **DRBG_Instantiate** function to obtain the entropy needed for instantiation. Let *counting_method* indicate the method for counting entropy from the entropy source(s) (i.e., Method 1 counts only entropy provided by physical entropy sources, and Method 2 counts entropy from non-physical and physical entropy sources; see Sec. 2.3).

1. When the DRBG is a CTR_DRBG without a derivation function, full-entropy bits **shall** be obtained from the entropy source(s) as follows:
    a. If all entropy sources provide full-entropy output, the **Get_randomness-source_input** call is replaced by:
        - (*status*, *seed_material*) = **Get_entropy_bitstring**(s + 128, *counting_method*).[16]
        - If (*status* ≠ SUCCESS), then return (*status, Invalid_state_handle*).

        The output of the entropy source(s) **shall** be concatenated to obtain the s + 128 full-entropy bits to be returned as *seed_material*.
    b. If one or more entropy sources do not provide full-entropy output, the **Get_randomness-source_input** call is replaced by: [17]
        - (*status*, *seed_material*) = **Get_conditioned_full_entropy_input**(s + 128, *counting_method*).
        - If (*status* ≠ SUCCESS), then return (*status, Invalid_state_handle*).
3. For CTR_DRBG (with a derivation function), Hash_DRBG, or HMAC_DRBG used as the DRBG, the entropy source(s) **shall** provide $3s/2$ bits of entropy to establish the security strength.

[15] If there is never more than one DRBG instantiation possible, then a state handle is not required.
[16] 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 800-90Ar1).
[17] See Sec. 3.2.2.2 for a specification of the **Get_conditioned_full_entropy_input** function.

a. If the implementer wants full entropy in the bitstring to be provided to the DRBG, the **Get_randomness-source_input** call is replaced by:

- (*status*, *seed_material*) = **Get_conditioned_full_entropy_input**(3*s*/2, *counting_method*).
- If (*status* ≠ SUCCESS), then return (*status, Invalid_state_handle*).

b. Otherwise, the **Get_randomness-source_input** call is replaced by either:

- (*status, seed material*) = **Get_entropy_bitstring**(3*s*/2, *counting_method*)

  OR

  (*status, seed_material*) = **Get_conditioned_ input**(3*s*/2, *counting_method*).
- If (*status* ≠ SUCCESS), then return (*status, Invalid_state_handle*).

5.2.2. Requesting Pseudorandom Bits From an RBG2 Construction

If prediction resistance is desired by a consuming application for the next RBG output to be generated so that previous internal states that may have been compromised cannot be used to determine the next RBG output, the application requests a reseed of the DRBG as discussed in Sec. 5.2.3 before requesting the generation of pseudorandom bits. Figure 20 depicts an (optional) reseed request before requesting the generation of pseudorandom bits.

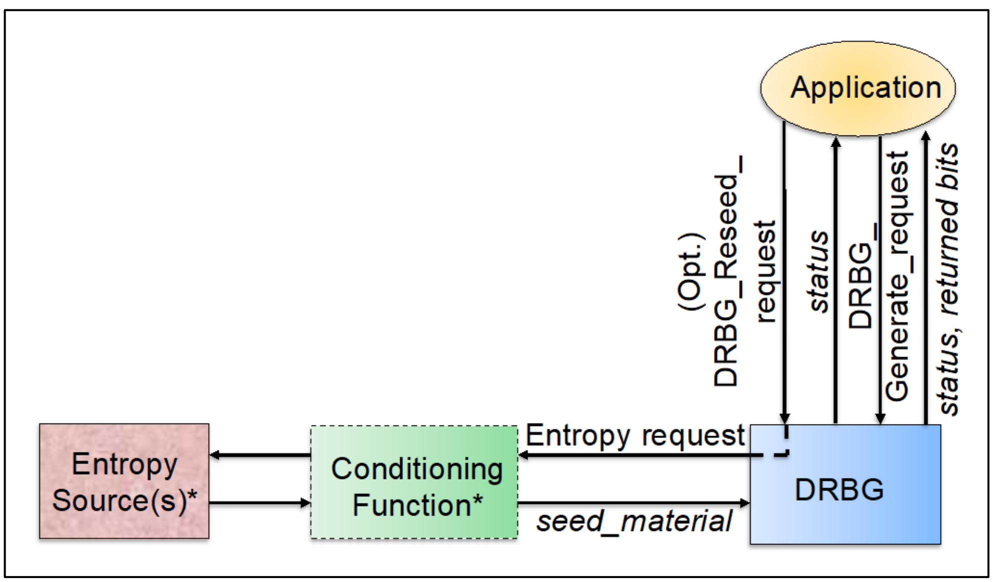

Fig. 20. RBG2 generate request following an optional reseed request

If a reseed of the RBG was not requested by the application prior to requesting the generation of pseudorandom bits or a *status* of SUCCESS was returned by the **DRBG_Reseed** function in response to a reseed request, pseudorandom bits are requested as follows (see Sec. 2.8.1.2):

(*status, returned_bits*) = **DRBG_Generate_request**(*RBG2_DRBG_state_handle, requested_number_of_bits, requested_security_strength, additional_input*).

The request **shall** result in the execution of a **DRBG_Generate** function by the DRBG (see Sec. 2.8.1.2) and checking the *status* returned by the **DRBG_Generate** function:

- (*status, returned_bits*) = **DRBG_Generate**(*RBG2_DRBG_state_handle, requested_number_of_bits, requested_security_strength, additional_input*).
- If (*status* ≠ SUCCESS), then return (*status, Null*).

The **DRBG_Generate** function returns the *status* of the process, which **shall** also be returned to the application in response to the **DRBG_Generate_request**. If the *status* indicates that the generation was successful, the requested random bits (*returned_bits*) are also provided by the **DRBG_Generate** function and forwarded to the application.

5.2.3. Reseeding an RBG2 Construction

The capability to reseed an RBG2 construction is optional. If implemented, the reseeding of the DRBG may be performed 1) upon request from a consuming application or 2) based on implementation-selected criteria, such as time, number of outputs, events, or the availability of sufficient entropy. The DRBG **should** be reseeded occasionally (e.g., after 2^{19} bits have been output).

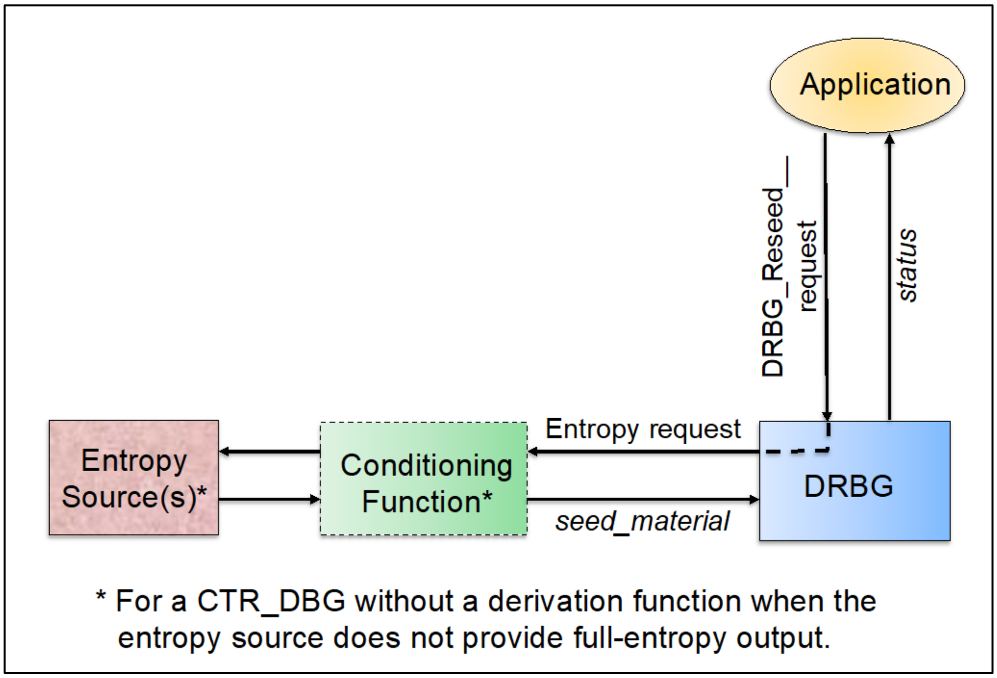

Fig. 21. Reseed request from an application

An application may request a reseed of the RBG2 construction (see Sec. 2.8.1.3):

status = **DRBG_Reseed_request**(*RBG2_DRBG_state_handle, additional_input*).

If the DRBG receives a **DRBG_Reseed_Request** or if the DRBG is scheduled for a reseed (see SP 800-90A), the **DRBG_Reseed** function **shall** be executed (see Sec. 2.8.1.3):

status = **DRBG_Reseed**(*RBG2_DRBG_state_handle, additional_input*).

The **DRBG_Reseed function** returns the *status* of the reseed process, which **shall** be returned to the application if requested using a **DRBG_Reseed_request**.

The **DRBG_Reseed** function uses a **Get_randomness-source_input** call to obtain the entropy needed for reseeding the DRBG (see Sec. 2.8.1.3 herein and SP 800-90A). 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 <u>without</u> a derivation function, use the appropriate replacement as specified in step 1 of Sec. 5.2.1.

2. For CTR_DRBG (with a derivation function), Hash_DRBG, or HMAC_DRBG, replace the **Get_randomness-source_input** call in the **DRBG_Reseed** function with the following:[18]

   a. If the implementer wants full entropy in the returned bitstring, the **Get_randomness-source_input** call is replaced by:

      (*status*, *seed_material*) = **Get_conditioned_full_entropy_input**(*s*, *counting_method*).

   b. Otherwise, the **Get_randomness-source_input** call is replaced by:

      (*status, seed_material*) = **Get_entropy_bitstring**(*s, counting_method*)

      OR

      (*status, seed_material*) = **Get_conditioned_ input**(*s, counting_method*).

5.3. RBG2 Construction Requirements

An RBG2 construction has the following requirements in addition to those specified in SP 800-90A and SP 800-90B:

1. The RBG **shall** employ an **approved** and validated DRBG from SP 800-90A 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 SP 800-90A, SP 800-90B, SP 800-90C, FIPS 140, and the specification of any other **approved** algorithm used within the RBG, as appropriate.
3. One or more validated entropy sources **shall** be used to instantiate and reseed the DRBG. A non-validated entropy source **shall not** be used for this purpose.
4. 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 source(s) used for instantiation (if a reseed capability is implemented).
5. 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(s) or from the entropy source(s) via an external vetted conditioning function that provides full-entropy output (see Sec. 3.2.2.2).
6. For CTR_DRBG (with a derivation function), Hash_DRBG, or HMAC_DRBG, a bitstring with at least $3s/2$ bits of entropy **shall** be obtained from the entropy source(s) 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(s). The

[18] See Sec. 2.8.2 and 3.1 for discussions of the **Get_entropy_bitstring** function.

entropy may be obtained directly from the entropy source(s) or via an external vetted conditioning function (see Sec. 3.2.2).

7. The entropy source(s) used for the instantiation and reseeding of the DRBG within 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 Sec. 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).

8. The entropy source(s) used for the instantiation and reseeding of the DRBG within 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 Sec. 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).

9. A specific entropy-source output (or portion thereof) **shall not** be reused (e.g., it is destroyed after use).

10. When a validated entropy source reports a failure, the failure **shall** be handled as discussed in item 10 of Sec. 2.6.

6. RBG3 Constructions Based on the Use of Physical Entropy Sources

An RBG3 construction is designed to provide full entropy (i.e., an RBG3 construction can support all security strengths). 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 RBG3 Description

The RBG3 constructions specified in this recommendation include one or more physical entropy sources and an **approved** DRBG from SP 800-90A. One or more non-physical entropy sources may optionally be included, but any entropy they provide is not counted. That is, Method 1 of Sec. 2.3 is used for counting entropy during RBG3 operation.

Upon receipt of a request for random bits from a consuming application, the RBG3 construction accesses its entropy source(s) to obtain sufficient bits for the request. See Sec. 3.1 for further discussion about accessing entropy sources.

An implementation may be designed so that the DRBG implementation used within an RBG3 construction can be directly accessed by a consuming application using the same internal state as the RBG3 construction. Access to the DRBG using a different internal state than is used by the RBG3 construction is allowed as specified in Sec. 5 without the additional restrictions imposed in Sec. 6.3, Requirement 3, and Sec. 6.5.2, Requirements 2 and 3.

The DRBG within an RBG3 construction is instantiated (i.e., seeded) at the highest security strength possible for its design (see Table 3). This is the fallback security strength if the entropy source fails in an undetected manner.

Table 3. Highest security strength for the DRBG's cryptographic primitive

Cryptographic Primitive	Highest Security Strength
AES-128	128
AES-192	192
AES-256	256
SHA-256/SHA3-256	256
SHA-384/SHA3-384	256
SHA-512/SHA3-512	256

If a failure of all physical entropy sources is detected, the RBG operation is terminated. Operation **must** be resumed only after repair and successful testing by instantiating the DRBG with new entropy from the entropy source(s).

If all physical entropy sources fail in an undetected manner, the RBG continues to operate as an RBG2(P) construction, providing outputs at the security strength instantiated for its DRBG (see Sec. 5). Although security strengths of 128 and 192 bits are allowed for the DRBG (depending on its cryptographic primitive), a DRBG that is capable of supporting a security strength of 256 bits and is instantiated at that strength is recommended so that the RBG will continue to operate at a security strength of 256 bits in the event of an undetected failure of the physical entropy source(s).

6.2. RBG3 Construction Types and Their Variants

Two basic 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 Sec. 6.4).
2. RBG3(RS) — This construction is based on using one or more validated entropy sources to continuously reseed the DRBG (see Sec. 6.5).

6.3. General Requirements

RBG3 constructions have the following general security requirements:

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 Sec. 2.3. Only the entropy provided by validated physical entropy sources **shall** be counted toward fulfilling entropy requests, although entropy provided by one or more validated non-physical entropy sources may be used but not counted.
2. An RBG3 construction and its components **shall** be successfully validated for compliance with the corresponding requirements in SP 800-90A, SP 800-90B, SP 800-90C, FIPS 140, and the specification of any other **approved** algorithm used within the RBG, as appropriate.
3. The DRBG **shall** be instantiated at its highest possible security strength before the first use of the RBG3 construction or direct access of the DRBG. A DRBG **should** support a security strength of 256 bits.
4. The RBG **shall** employ an **approved** and validated DRBG from SP 800-90A whose highest possible security strength is the targeted fallback security strength for the DRBG (see Sec. 6.1).
5. A specific entropy-source output (or portion thereof) **shall not** be reused (e.g., the same entropy-source output **shall not** be used for an RBG3 request or for seeding or reseeding the DRBG).
6. If the DRBG is directly accessible, the requirements in Sec. 5.3 for RBG2(P) constructions **shall** apply to the direct access of the DRBG.
7. If a failure is detected within the RBG, see Sec. 2.6 (item 10) and 3.1.

See Sec. 6.4.2 and 6.5.2 for additional requirements for the RBG3(XOR) and RBG3(RS) constructions, respectively.

6.4. 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 during the generate process (see Fig. 22).

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 Sec. 2.1). If the entropy source(s) cannot provide full-entropy output, then an external conditioning function **shall** be used to condition the output of the entropy source(s) to a full-entropy bitstring before XORing with the output of the DRBG (see Sec. 3.2.2.2).

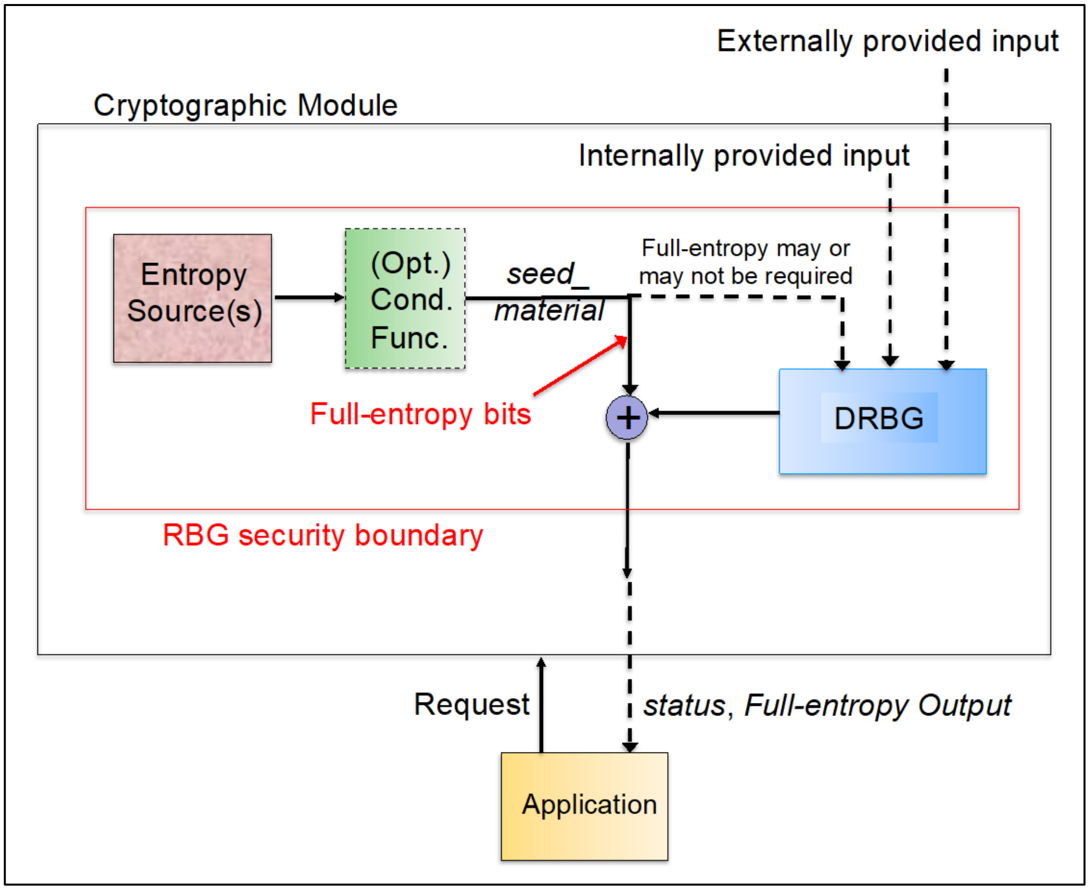

Fig. 22. Generic structure of the RBG3(XOR) construction

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(s) or from the combination of validated entropy sources and an external vetted conditioning function that provides full-entropy output (see Sec. 3.2.2.2). When the entropy sources are working properly,[19] an n-bit output from the RBG3(XOR) construction is said to provide n bits of entropy

[19] The entropy source(s) provide(s) at least the amount of entropy determined during the entropy-source validation process.

or to support a security strength of *n* bits. An example of an RBG3(XOR) design is provided in Appendix B.5.

6.4.1. Conceptual Interfaces

The RBG interfaces include function calls for instantiating the DRBG (see Sec. 6.4.1.1), generating random bits on request (see Sec. 6.4.1.2), and reseeding the DRBG instantiation (see Sec. 6.4.1.3).

6.4.1.1. Instantiation of the DRBG

As discussed in Sec. 2.8.3.1, before the RBG3(XOR) construction can be used to generate bits, an application instantiates the DRBG within the construction:

(*status, state_handle*) = **Instantiate_RBG3_DRBG_request**(*requested_security_strength, personalization_string*),

where *requested_security_strength* and *personalization_string* are optional. If the *requested_security_strength* parameter is provided and exceeds the highest security strength that can be supported by the DRBG, an error indication **shall** be returned with an invalid *state_handle* (see Sec. 2.8.3.1).

If the *requested_security_strength* is provided and is acceptable (i.e., *requested_security_strength* does not exceed the highest security strength that can be supported by the DRBG; see Sec. 2.8.3.1) or if the *requested_security_strength* parameter is not provided, the **Instantiate_RBG3_DRBG_request** received by the RBG3(XOR) construction **shall** result in the execution of the **RBG3(XOR)_Instantiate** function below. The *status* returned by the **RBG3(XOR)_Instantiate** function **shall** be returned to the application in response to the **Instantiate_RBG3_DRBG_request**. The return of the *state_handle* is optional if only a single instantiation is allowed by an implementation.

Let *s* be the highest security strength that can be supported by the DRBG. The DRBG in the RBG3(XOR) construction is instantiated as follows:

RBG3(XOR)_Instantiate:

Input:

1. *s*: The security strength to be instantiated for the DRBG.
2. *personalization_string*: An optional (but recommended) personalization string.

Output:

1. *status*: The status returned by the **RBG3(XOR)_Instantiate** function.
2. *RBG3_DRBG_state_handle*: The returned state handle for the internal state of the DRBG or an invalid state handle.

Process:

1. (*status, RBG3_DRBG_state_handle*) = **DRBG_Instantiate**(*s, personalization_string*).
2. If (*status* ≠ SUCCESS), then return (*status, Invalid_state_handle*).
3. Return (SUCCESS, *RBG3_DRBG_state_handle*).

In step 1, the DRBG is instantiated at a security strength of *s* bits. *RBG3_DRBG_state_handle* (if returned) is the state handle for the internal state of the DRBG used within the RBG3(XOR) construction.

In step 2, if the *status* returned from step 1 does not indicate a success, then return the *status* with an invalid state handle.

In step 3, the *status* and *RBG3_DRBG_state_handle* that were obtained in step 1 are returned to the requesting application.

The handling of status codes is discussed in item 10 of Sec. 2.6 and in Sec. 2.8.3, 3.1, and 8.1.2.

6.4.1.2. Random Bit Generation by the RBG3(XOR) Construction

As discussed in Sec. 2.8.3.2, an application may request the generation of random bits from the RBG3(XOR) construction:

(*status, returned_bits*) = **RBG3_DRBG_Generate_request**(*RBG3_DRBG_state_handle, n, additional_input*),

where *RBG3_DRBG_state_handle* was provided during instantiation (see Sec. 6.4.1.1), *n* is the number of bits to be generated and returned to the application, and *additional_input* is optional.

The **RBG3_DRBG_Generate_request** received by the RBG3(XOR) construction **shall** result in the execution of the **RBG3(XOR)_Generate** function below. The output of that function **shall** be returned to the application in response to the **RBG3_DRBG_Generate_request**.

Let *s* be the security strength instantiated for the DRBG (i.e., the highest security strength that can be supported by the DRBG; see Sec. 6.4.1.1), and let the *RBG3_DRBG_state_handle* be the value returned by the instantiation function for RBG3(XOR)'s DRBG instantiation. Random bits with full entropy **shall** be generated by the RBG3(XOR) construction using the following generate function with the values of *n* and *additional_input* provided in the **DRBG_Generate_request** as input:

RBG3(XOR)_Generate:

Input:

1. *RBG3_DRBG_state_handle*: The state handle of the DRBG used by the RBG3 construction.
2. *n*: The number of bits to be generated.

3. *additional_input*: Optional additional input.

Output:

1. *status*: The status returned by the **RBG3(XOR)_Generate** function.
2. *returned_bits*: The *n* bits generated by the RBG3(XOR) construction or a *Null* string.

Process:

1. (*status*, *ES_bits*) = **Request_entropy**(*n*). (See the notes below for customizing this step.)
2. If (*status* ≠ SUCCESS), then return (*status*, *Null*).
3. (*status*, *DRBG_bits*) = **DRBG_Generate**(*RBG3_DRBG_state_handle, n, s, additional_input*).
4. If (*status* ≠ SUCCESS), then return (*status*, *Null*).
5. *returned_bits* = *ES_bits* ⊕ *DRBG_bits*.
6. Return (SUCCESS, *returned_bits*).

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

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

  (*status*, *ES_bits*) = **Get_entropy_bitstring**(*n*, *Method_1*).

  The **Get_entropy_bitstring** function[20] **shall** use Method 1 in Sec. 2.3 to obtain the *n* full-entropy bits that were requested to produce *ES-bits*.

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

  (*status*, *ES_bits*) = **Get_conditioned_full_entopy_input**(*n*, *Method_1*).

  The **Get_conditioned_full_entropy_input** procedure is specified in Sec. 3.2.2.2. It requests entropy from the entropy sources in step 3.1 of that procedure with a **Get_entropy_bitstring** call. The **Get_entropy_bitstring** call **shall** use Method 1 (as specified in Sec. 2.3) when collecting the output of the entropy source(s) (i.e., only the entropy provided by one or more physical entropy sources are 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 a *Null* bitstring as the *returned_bits*. If *status* indicates a success, *ES_bits* is the full-entropy bitstring to be used in step 5.

[20] See Sec. 2.8.2 and 3.2.

In step 3, the RBG3(XOR)'s DRBG instantiation is requested to generate *n* bits at a security strength of *s* bits. The DRBG instantiation is indicated by the *RBG3_DRBG_state_handle*, which was obtained during instantiation (see Sec. 6.4.1.1). If additional input is provided in the **RBG3(XOR)_Generate** call, it **shall** be included in the **DRBG_Generate** function call to the DRBG. It is possible that the DRBG may require reseeding during the **DRBG_Generate** function call in step 3 (e.g., because the end of the seedlife of the DRBG has been reached).

In step 4, if the **DRBG_Generate** function request is not successful, the **RBG3(XOR)_Generate** function is aborted, and the *status* received in step 3 and 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 source(s) (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.

6.4.1.3. Pseudorandom Bit Generation Using a Directly Accessible DRBG

If prediction resistance is desired by a consuming application for the next DRBG output to be generated so that a previous internal state that may have been compromised cannot be used to determine the next DRBG output, the application requests a reseed of the DRBG before requesting the generation of pseudorandom bits directly from the DRBG, as discussed in Sec. 6.4.1.4. This is the same process shown in Fig. 20 in Sec. 5.2.2.

If a reseed of the DRBG was not requested by the application, or a *status* of SUCCESS was returned by the **DRBG_Reseed** function when the application requested a reseed, pseudorandom bits may be requested as follows:

(*status, returned_bits*) = **DRBG_Generate_request**(*RBG3(XOR)_DRBG_state_handle, requested_number_of_bits, requested_security_strength, additional_input*),

where *RBG3(XOR)_state_handle* was provided during instantiation and *additional_input* is optional.

The **DRBG_Generate_request** received by the DRBG **shall** result in the execution of the **DRBG_Generate** function in the DRBG:

(*status, returned_bits*) = **DRBG_Generate**(*RBG3_DRBG_state_handle, requested_number_of_bits, requested_security_strength, additional_input*),

where:

- *RBG3_DRBG_state_handle* is the state handle used by the DRBG within the RBG3(XOR) construction.
- *requested_security_strength* is provided in the **DRBG_Generate_request** and must be ≤ the instantiated security strength of the DRBG.

- Any *additional_input* provided in a **DRBG_Generate_request shall** be provided as input to the **DRBG_Generate** function. Otherwise, the use of *additional_input* is optional.

The output of the **DRBG_Generate** function **shall** be returned to the application in response to the **DRBG_Generate_request**.

6.4.1.4. Reseeding the DRBG Instantiation

As discussed in Sec. 2.4.2, the reseeding of the DRBG may be performed 1) upon request from a consuming application or 2) based on implementation-selected criteria, such as time, number of outputs, events, or the availability of sufficient entropy.

An application may request the reseeding of the DRBG within the RBG3(XOR) construction:

status = **DRBG_Reseed_request**(*RBG3(XOR)_DRBG_state_handle, additional_input*),

where *RBG3(XOR)_state_handle* was provided during instantiation and *additional_input* is optional.

The DRBG executes a **DRBG_Reseed** function in response to a **DRBG_Reseed_request** from an application or in accordance with implementation-selected criteria:

status = **DRBG_Reseed**(*RBG3_DRBG_state_handle, additional_input*),

where *RBG3_DRBG_ state_handle* (if used) was returned by the **DRBG_Instantiate** function (see Sec. 2.8.1.1 and 6.4.1.1). *RBG3_DRBG_state_handle* is the state handle for the internal state of the DRBG within the RBG3(XOR) construction. Any *additional_input* provided in a **DRBG_Reseed_request shall** be provided as input to the **DRBG_Reseed** function. Otherwise, the use of *additional_input* is optional.

6.4.2. RBG3(XOR) Requirements

An RBG3(XOR) construction has the following requirements in addition to those provided in Sec. 6.3:

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 that provides full-entropy output using the output of one or more validated physical entropy sources.
2. Entropy source output used for the RBG's XOR operation **shall not** also be used to instantiate and reseed the RBG's DRBG.[21]
3. The DRBG instantiation **should** be reseeded occasionally (e.g., after a predetermined period of time or number of generation requests).

[21] However, the same entropy source(s) may be used to provide entropy for the XOR operation and to seed and reseed the RBG's DRBG.

6.5. RBG3(RS) Construction

The second RBG3 construction specified in this document is the RBG3(RS) construction shown in Fig. 23. An example of this construction is provided in Appendix B.6.

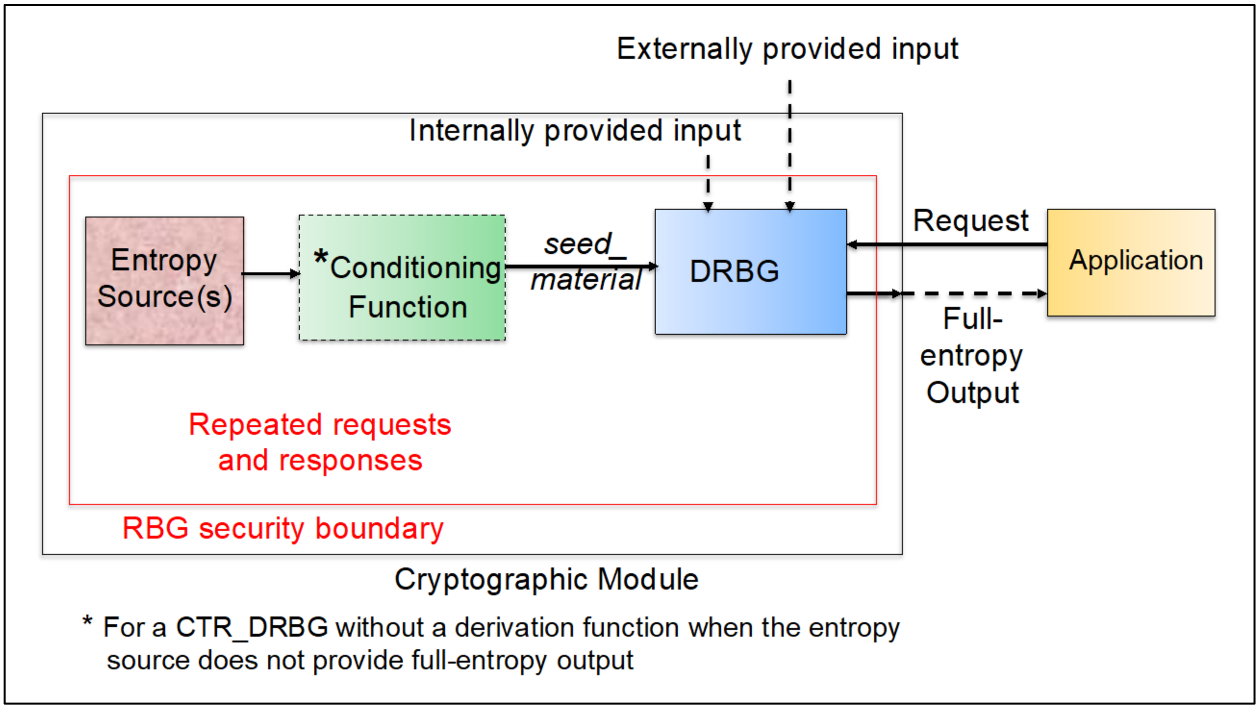

Fig. 23. Generic structure of the RBG3(RS) construction

External conditioning of the outputs from the entropy source(s) during instantiation and reseeding is required to provide bitstrings with full entropy when the DRBG is a CTR_DRBG without a derivation function and the entropy source(s) do not provide output with full entropy. Otherwise, the use of a conditioning function is optional.

6.5.1. Conceptual Interfaces

The RBG interfaces include function calls for instantiating the DRBG (see Sec. 6.5.1.1), generating random bits on request (see Sec. 6.5.1.2), and reseeding the DRBG instantiation (see Sec. 6.5.1.3).

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

Before the RBG3(RS) construction can be used to generate bits, an application **shall** request the instantiation of the DRBG within the construction (see Sec. 2.8.3.1):

(*status*, *RBG3_DRBG_state_handle*) =
Instantiate_RBG3_DRBG_request(*requested_security_strength*, *personalization_string*),

where *requested_security_strength* and *personalization_string* are optional. If the *requested_security_strength* parameter is provided and exceeds the highest security strength that can be supported by the DRBG design, an error indication **shall** be returned with an invalid *state_handle* (see Sec. 2.8.3.1).

If the *requested_security_strength* is provided and acceptable (see Sec. 2.8.3.1) or the *requested_security_strength* information is not provided, the **Instantiate_RBG3_DRBG_request** received by the RBG3(RS) construction **shall** result in the execution of the **RBG3(RS)_Instantiate** function below. The *status* returned by that function **shall** be returned to the application in response to the **Instantiate_RBG3_DRBG_request**.

Let *s* be the highest security strength that can be supported by the DRBG, and let *personalization_string* be the value provided in the **Instantiate_RBG3_DRBG_request** (if any). The DRBG in the RBG3(RS) construction is instantiated as follows:

RBG3(RS)_Instantiate:

Input:

1. *s*: The requested security strength for the DRBG in the RBG3(RS) construction.
2. *personalization_string*: An optional (but recommended) personalization string.

Output:

1. *status*: The status returned from the **RBG3(RS)_Instantiate** function.
2. *RBG3_DRBG_state_handle*: A pointer to the internal state of the DRBG if the *status* indicates a success. Otherwise, an invalid state handle.

Process:

1. (*status, RBG3_DRBG_state_handle*) = **DRBG_Instantiate**(*s, personalization_string*).
2. If (*status* ≠ SUCCESS), then return (*status*, *Invalid_state_handle*).
3. Return (SUCCESS, *RBG3_DRBG_state_handle*).

In step 1, the DRBG is instantiated at a security strength of *s* bits.

In step 2, if the *status* returned from step 1 does not indicate a success, then return the *status* and an invalid state handle.

In step 3, the *status* and the *RBG3_DRBG_state_handle* are returned. *RBG3_DRBG_state_handle* is the state handle for the internal state of the DRBG used within the RBG3(RS) construction.

The handling of status codes is discussed in Sec. 2.8.3 and 6.5.1.2.

6.5.1.2. Random and Pseudorandom Bit Generation

6.5.1.2.1. Generation Using the RBG3(RS) Construction

When the DRBG within an RBG3(RS) construction is instantiated at a security strength of s bits, s bits with full entropy can be extracted from its output if at least $s + 64$ bits of fresh entropy are inserted into the DRBG's internal state before generating the output (see item 11 in Sec. 2.6). Per requirement 4 in Sec. 6.3, the security strength and the resulting length of the full-entropy bitstring (s) is the highest security strength possible for the cryptographic primitive used by the DRBG. If a consuming application requests more than s bits, multiple iterations of this process are required.

Figure 24 depicts a sequence of RBG3(RS) generate operations. Full-entropy output from this construction is generated in s-bit strings, where s is the instantiated security strength of the DRBG used in an implementation. For each s bits of generated output, $s + 64$ bits of fresh entropy are obtained by reseeding (shown in red in the figure) and then inserted into the DRBG's internal state before generating an s-bit string (shown in blue). Two generate requests using the RBG3(RS) construction are shown in the figure. The first generate request requires the generation of two iterations of the reseed-generate process (i.e., two strings of s bits are generated, each preceded by obtaining $s + 64$ bits of fresh entropy). The second generate request requires only a single string of s full-entropy bits to be generated (preceded by obtaining $s + 64$ bits of fresh entropy).

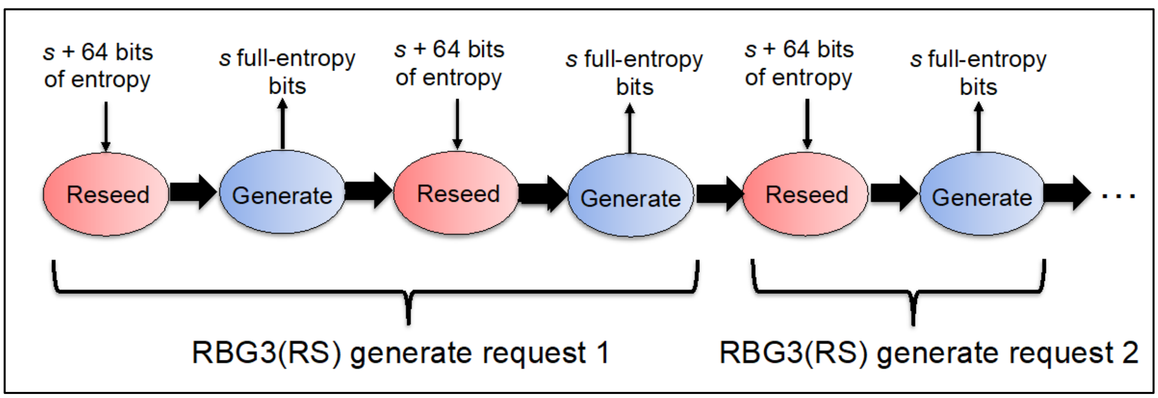

Fig. 24. Sequence of RBG3(RS) generate requests

Figure 25 provides a flow of the steps of the **RBG3(RS)_Generate** function.

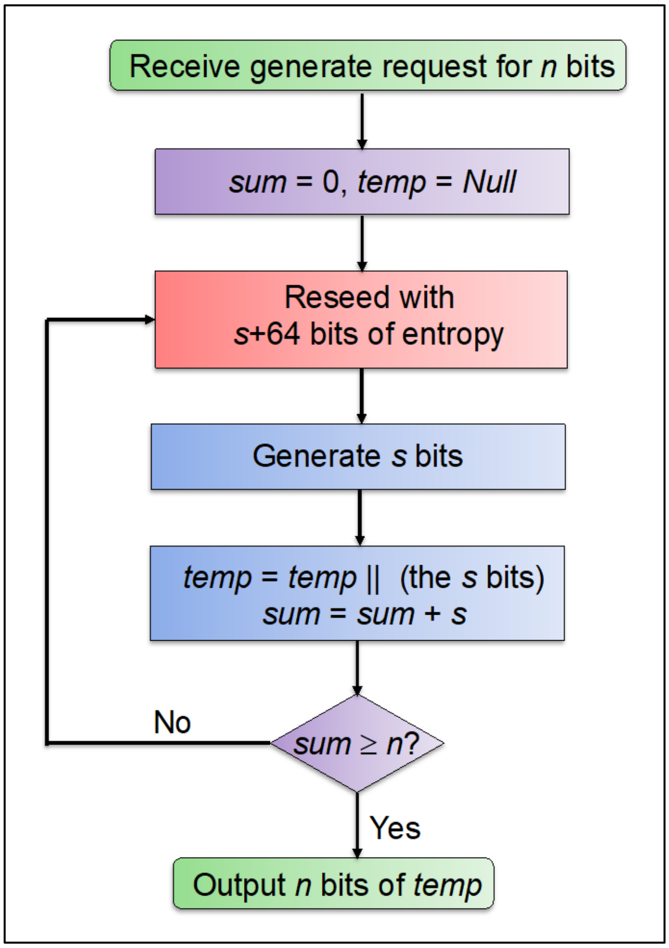

Fig. 25. Flow of the RBG3(RS)_Generate function

Figure 26 depicts a sequence of RBG3(RS) generate requests followed by a sequence of requests directly to the DRBG (shown in green) and another sequence of RBG3(RS) generate requests. As previously discussed, an RBG3(RS) generate request is preceded by obtaining $s + 64$ bits of fresh entropy. The first generate request directly to the DRBG following one or more RBG3(RS) generate requests is preceded by obtaining $s + 64$ bits of fresh entropy. Successive DRBG requests do not require the insertion of fresh entropy (except, for example, if requested by the consuming application). When a consuming application later requests that the RBG3(RS) construction generate full-entropy bits again, the reseed-generate process is resumed by first reseeding with $s + 64$ bits of entropy before the generation of each s-bit string by the RBG3(RS) construction.

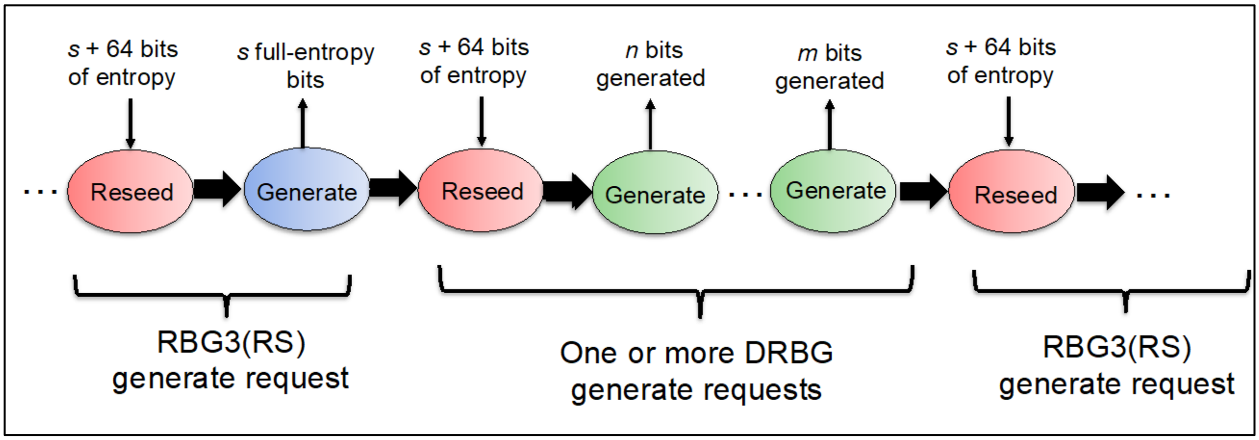

Fig. 26. Direct DRBG generate requests

As discussed in Sec. 2.8.3.2, an application may request the generation of random bits as follows:

(*status, returned_bits*) = **RBG3_ Generate_request**(*RBG3_DRBG_state_handle, n, additional_input*),

where *RBG3_DRBG_state_handle* was provided during instantiation (see Sec. 6.5.1.1), *n* is the number of bits to be generated and returned to the application, and *additional_input* is optional.

The **RBG3_Generate_request** received by the RBG3(RS) construction **shall** result in the execution of the **RBG3(RS)_Generate** function below. The output of that function **shall** be returned to the application in response to the **RBG3_DRBG_Generate_request**.

Let the input parameters provided in the request above also be provided as input to the **RBG3(RS)_Generate** function. Appendix A.2 is a reference for the appropriate values for each DRBG type.

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

RBG3(RS)_ Generate:

Input:

1. *RBG3_DRBG_state_handle*: A pointer to the internal state of the DRBG used by the RBG3(RS) construction.
2. *n*: The number of full-entropy bits to be generated.
3. *additional_input*: Optional additional input.

Output:

1. *status*: The status returned by the **RBG3(RS)_Generate** function.
2. *returned_bits*: The *n* full-entropy bits requested or a *Null* string.

Process:

1. *temp* = *Null*.

2. *sum* = *0*.
3. While (*sum* < *n*),
   3.1 Reseed with at least $s + 64$ bits of fresh entropy (see the notes below for customizing this step).
   3.2 (*status*, *full_entropy_bits*) = **DRBG_Generate**(*RBG3_DRBG_state_handle*, *s*, *s*, *additional_input*).
   3.3 If (*status* ≠ SUCCESS), then return (*status*, *Null*).
   3.4 *temp* = *temp* || *full_entropy_bits*.
   3.5 *sum* = *sum* + *s*.
   3.6 *additional_input* = *Null* string.
4. Return (SUCCESS, **leftmost**(*temp*, *n*)).

In steps 1 and 2, the bitstring intended to collect the generated bits (*temp*) is initialized to the *Null* bitstring, and the counter for the number of bits obtained for fulfilling the request (*sum*) is initialized to zero.

Step 3 is iterated until at least *n* full-entropy bits have been generated.

Step 3.1 obtains at least $s + 64$ bits of fresh entropy and inserts it into the internal state.

- For CTR_DRBG <u>without</u> a derivation function, $s + 128$ bits of entropy are requested during reseeding using a randomness source that provides full-entropy output. Step 3.1 becomes:
    - *status* = **DRBG_Reseed**(*RBG3_DRBG_state_handle, additional_input*).
    - If (*status* ≠ SUCCESS), then return (*status, Null*)

  with the **Get_randomness-source_input** call in the **DRBG_Reseed** function replaced by:
    - (*status*, *seed_material*) = **Get_entropy_bitstring**($s + 128$, *Method_1*).
    - If (*status* ≠ SUCCESS), then return (*status*, *Null*),

  where *Method_1* indicates that only the entropy from physical entropy sources is counted.
- For a Hash_DRBG, HMAC_DRBG, or CTR_DRBG <u>with</u> a derivation function, *s* bits of fresh entropy are usually inserted into the internal state during a **DRBG_Reseed** function. To insert $s + 64$ bits into the internal state, two methods are provided:

  <u>Method A</u> is a modification of the **DRBG_Reseed** function that requests $s + 64$ bits of entropy from the entropy source(s) rather than (the usual) *s* bits (see Fig. 27). Making this change is straightforward, given access to the internals of a DRBG implementation.

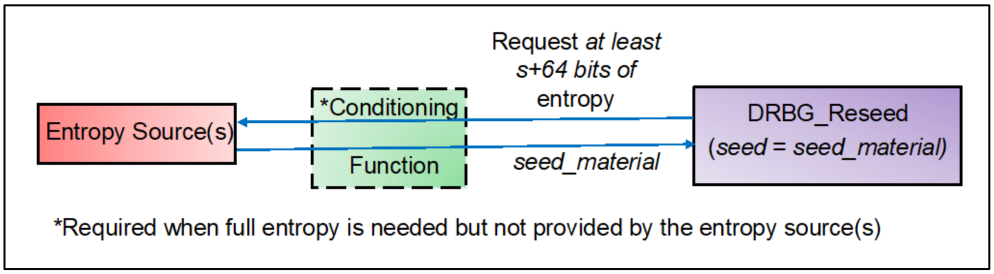

Fig. 27. Modification of the DRBG_Reseed function

Step 3.1 becomes:

- *status* = **DRBG_Reseed**(*RBG3_DRBG_state_handle, additional_input*)
- If (*status* ≠ SUCCESS), then return (*status, Null*)

with the **Get_randomness-source_input** call in the **DRBG_Reseed** function replaced by:

- (*status*, *seed_material*) = **Get_entropy_bitstring**(*s* + 64, *Method_1*).
- If (*status* ≠ SUCCESS), then return (*status, Null*).

Method_1 indicates that only the entropy from physical entropy sources is to be counted.

Method B (depicted in Fig. 28) first obtains a bitstring with 64 bits of entropy directly from the entropy source(s). It then invokes the **DRBG_Reseed** function using this bitstring as additional input (called *extra_bits* below to avoid confusion with the *additional_input* provided by the application when invoking the **DRBG_Generate_request** above). The **DRBG_Reseed** function will obtain *s* bits of entropy from the entropy source(s),[22] combine it with the 64 bits of entropy provided as the *extra_bits* and incorporate the result into the DRBG's internal state. This method is appropriate when the RBG3(RS) construction is being implemented using an existing DRBG implementation that cannot be altered.

[22] The value of *s* is recorded in the DRBG's internal state (see SP 800-90A).

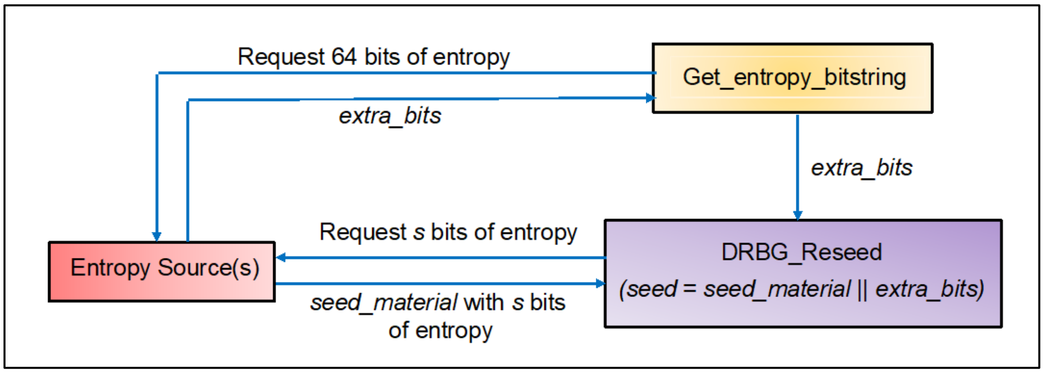

Fig. 28. Request extra bits before reseeding

Step 3.1 becomes:

3.1.1 (*status*, *extra_bits*) = **Get_entropy_bitstring**(64, *Method_1*).

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

3.1.3 *status* = **DRBG_Reseed**(*RBG3_DRBG_state_handle, extra_bits* || *additional_input*).

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

In step 3.1.3, the **Get_randomness-source_input** call in the **DRBG_Reseed** function is replaced by:

- (*status*, *seed_material*) = **Get_entropy_bitstring**(*s*, *Method_1*).
- If (*status* ≠ SUCCESS), then return (*status, Null*).

Method_1 indicates that only the entropy from physical entropy sources is to be counted.

In step 3.2, request the generation of *full_entropy_bits* using the **DRBG_Generate** function, where:

- The *RBG3_DRBG_state_handle* was obtained during DRBG instantiation (see Sec. 6.5.1.1).
- *s* is both the number of full-entropy bits to be produced during the **DRBG_Generate** function call and the security strength of the DRBG instantiation (see Sec. 2.8.1.2 and Table 4 in Appendix A.2).
- *additional_input* is the current value of the *additional_input* string (initially provided in the **DRBG_Generate** call, used in the first iteration of step 3.2, and subsequently set to the *Null* string in step 3.6).

In step 3.3, if step 3.2 returned a *status* value indicating that the **DRBG_Generate** function was not successful, then return the *status* to the calling application with a *Null* bitstring.

In step 3.4, concatenate the *full_entropy_bits* obtained in step 3.2 to the temporary bitstring (*temp*).

In step 3.5, increment the output-length counter (*sum*) by *s* bits (i.e., the number of full-entropy bits obtained in step 3.2).

In step 3.6, to avoid reusing the *additional_input*, set its value to a *Null* string for subsequent iterations of step 3.

If $sum < n$, go to step 3.1.

Step 4 returns a *status* indicating SUCCESS to the calling application along with the leftmost *n* bits of *temp* as the *returned_bitstring*.

6.5.1.2.2. Generation Using a Directly Accessible DRBG

As discussed in Sec. 2.8.1.2, the DRBG used by the RBG3(RS) construction may be requested to generate output directly using the following request:

(*status, returned_bits*) = **DRBG_Generate_request**(*RBG3_DRBG_state_handle, requested_number_of_bits, requested_security_strength, additional_input*),

where *RBG3_DRBG_state_handle* was provided during instantiation (see Sec. 6.5.1.1) and *additional_input* is optional.

Before generating the requested output, the DRBG needs to be reseeded in the following circumstances:

1. Accessing a DRBG directly to generate output by the DRBG in the RBG3(RS) construction requires that the DRBG be reseeded with at least $s + 64$ bits of entropy from the entropy source(s) when the DRBG was previously used as a component of the **RBG3(RS)_generate** function. This requires that the RBG3(RS) implementation keep track of the type of generate request that was made previously (e.g., including this information in the DRBG's internal state) so that the reseeding of the DRBG is automatically performed before generating the requested DRBG output.
2. During a sequence of generate requests, the DRBG may reseed itself in response to some event.

Reseeding is accomplished as specified in Sec. 6.5.1.3.

If a reseed of the DRBG was not performed or a *status* of SUCCESS was returned by the **DRBG_Reseed** function when performed under conditions 1 or 2 above, the **DRBG_Generate_request** invokes the **DRBG_Generate** function (see Sec. 5.2.2), obtains the *status* of the operation and any generated bits (i.e., *returned_bits*), and forwards them to the application in response to the **DRBG_Generate_request**.

6.5.1.3. Reseeding

Reseeding the DRBG may be performed:

1. When explicitly requested by the consuming application,

2. During an **RBG3(RS)_generate** request (see Sec. 6.5.1.2.1) or in response to a direct DRBG generate request when the previous use of the DRBG was as a component of the **RBG3(RS)_Generate** function (see Sec. 6.5.1.2.2), or
3. Based on implementation-selected criteria, such as time, number of outputs, events, or the availability of sufficient entropy.

Case 1: An application sends a reseed request to the RBG:

status = **DRBG_Reseed_request**(*RBG3_DRBG_state_handle, additional_input*),

where *RBG3_DRBG_state_handle* was obtained during instantiation (see Sec. 6.5.1.1) and *additional_input* is optional.

Any *additional_input* provided by a **DRBG_Reseed request** from the application **shall** be used as input to the **DRBG_Reseed** function. Otherwise, the use of *additional_input* is optional.

The **DRBG_Reseed_request** results in the invocation of the **DRBG_Reseed** function (see Sec. 5.2.3). The *status* returned from the **DRBG_Reseed** function is forwarded to the application in response to the **DRBG_Reseed_request**.

Case 2: The DRBG is reseeded as follows:

- For CTR_DRBG without a derivation function, $s + 128$ bits of entropy are requested during reseeding in the same manner as for instantiation (see step 3.1 of Sec. 6.5.1.2.1).
- For a Hash_DRBG, HMAC_DRBG, or CTR_DRBG with a derivation function, use Method A or Method B (as specified in step 3.1 of Sec. 6.5.1.2.1) to obtain $s + 64$ bits of fresh entropy in the DRBG.

Case 3: A reseed of the DRBG is invoked based on implementation-selected criteria:

status = **DRBG_Reseed**(*RBG3_DRBG_state_handle, additional_input*).

For a CTR_DRBG, the DRBG is reseeded with $s + 128$ bits of fresh entropy. Otherwise, the DRBG is reseeded with either s or $s + 64$ bits of fresh entropy, depending on whether Method A or Method B was used in step 3.1 of Sec. 6.5.1.2.1.

6.5.2. Requirements for an RBG3(RS) Construction

An RBG3(RS) construction has the following requirements in addition to those provided in Sec. 6.3:

1. For each s bits generated by the RBG3(RS) construction, $s + 64$ bits of fresh entropy **shall** be acquired either directly from independent, validated entropy sources or from an external conditioning function that processes the output of the validated entropy sources to provide full-entropy, as specified in Sec. 3.2.2.2.

2. If the DRBG is directly accessible and the previous use of the DRBG was by the RBG3(RS) construction, a reseed of the DRBG instantiation with at least $s + 64$ bits of entropy **shall** be performed before generating output.
3. The DRBG **shall** be reseeded in accordance with Sec. 6.5.1.3.

7. RBGC Construction for DRBG Chains

The RBGC construction allows the use of a chain of DRBGs in which one DRBG is used to provide seed material for another DRBG. This design is common on many computing platforms and allows some level of modularity (e.g., an operating system RBG can be designed and validated without knowing the randomness source that will be available on the particular hardware on which it will be used, or a software application can be designed with its own RBG but without knowing the operating system or hardware used by the application).

7.1. RBGC Description

7.1.1. RBGC Environment

Figure 29 depicts RBGC constructions and the environment in which they will be used. An RBGC construction consists of an **approved** DRBG mechanism (from SP 800-90A) and the randomness source used for seeding and (optional) reseeding. This figure illustrates a tree of RBGC constructions that consists of two DRBG chains: 1) a chain consisting of $DRBG_1$, $DRBG_2$, and $DRBG_3$ and 2) a chain consisting of $DRBG_1$ and $DRBG_4$.

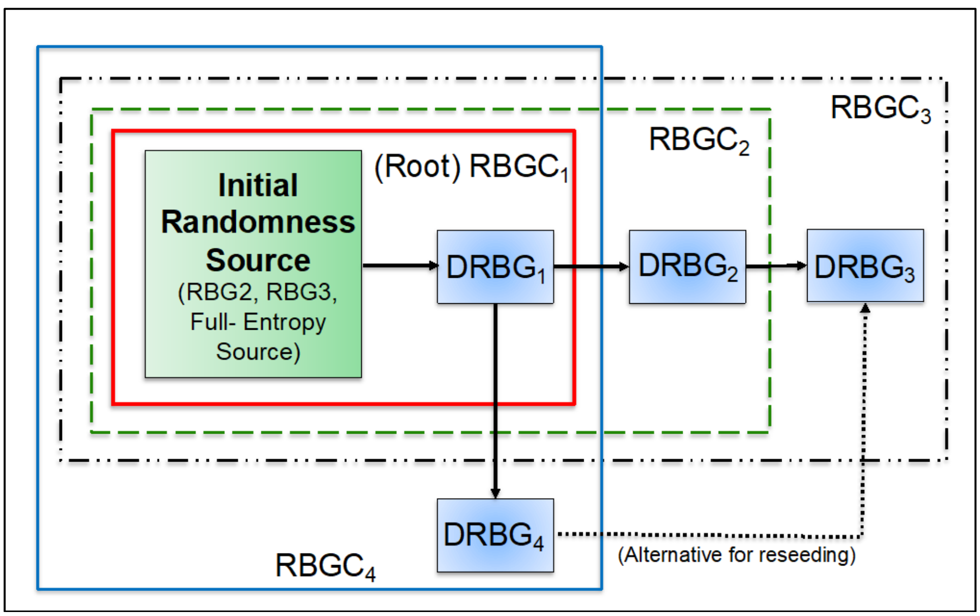

Fig. 29. DRBG tree using the RBGC construction

The core of this type of construction is called the *root* and is shown as $RBGC_1$ within the solid red rectangle in the figure. Its DRBG is labeled as $DRBG_1$, and its randomness source for seeding and (optionally) reseeding is labeled as the *initial randomness source*.

For each of the other RBGC constructions (i.e., RBG_2, RBG_3, and RBG_4), the DRBG within the construction is seeded by a DRBG within a "parent" RBGC construction (i.e., the parent is the randomness source used for seeding the DRBG). For $RBGC_2$ (shown as a box outlined with long green dashes [– – –]), the parent randomness source is the root (i.e., $RBGC_1$). For $RBGC_3$ (shown as a box with black dashes and dots [– • • – • • –]), the parent randomness source is $RBGC_2$. For $RBGC_4$ (shown within a box outlined with a solid blue rectangle), the parent randomness source is $RBGC_1$ (i.e., the root).

An RBGC construction may be used to Instantiate and reseed other RBGC constructions or to provide output for one or more applications (not shown in Fig. 29). All components of an RBGC tree — including the initial randomness source and the DRBG chains in that tree — reside on the same computing platform. The initial randomness source is not physically removable while the computing platform is operational, and the contents of the internal state of any DRBG in the tree are never relocated to another computer platform or output for external storage. See Appendix A.3 for a discussion about the intended meaning of a computing platform and implementation considerations.

Each RBGC construction may be a *parent* for one or more *child* RBGC constructions. Each of the child RBGC constructions has only one parent that serves as its randomness source for seeding the DRBG within it. Using Fig. 29 as an example, $RBGC_1$ is the only parent of both $RBGC_2$ and $RBGC_4$. $RBGC_2$ is the randomness source (i.e., the only parent) of $RBGC_3$. However, the parent may have siblings that may be used for reseeding under certain conditions (see Sec. 7.1.2.1) if the parent is not available to do so (e.g., the RBGC construction has been moved to a different core). In Fig. 29, $RBGC_2$ and $RBGC_4$ are siblings since they have the same parent ($RBGC_1$). In this case, the alternative path for reseeding is shown as a line of black dots.

An RBGC construction cannot have itself as a predecessor randomness source for reseeding. That is, there are no "seed loops" in which an RBGC construction provides seed material for a predecessor RBGC construction (e.g., a parent or grandparent). For example, in Fig. 29, $RBGC_2$ can be used as the randomness source for $RBGC_3$, but $RBGC_3$ cannot be used as the randomness source for reseeding $RBGC_1$ or $RBGC_2$. However, *additional_input* provided to the DRBG during a reseed or generate request may be anything, including the output of any RBGC construction of the tree.

7.1.2. Instantiating and Reseeding Strategy

7.1.2.1. Instantiating and Reseeding the Root RBGC Construction

The root RBGC construction is instantiated and (optionally) reseeded using an initial randomness source, which is either a validated full-entropy source or a validated RBG2(P), RBG2(NP), RBG3(XOR), or RBG3(RS) construction. An RBG2(P) or RBG2(NP) construction used as the initial randomness source **shall** have a capability of being reseeded on demand by the root.[23] A validated full-entropy source is a validated entropy source that provides full-entropy output or the combination of a validated entropy source and an external vetted conditioning function that

[23] A reseed of the initial randomness source is required for instantiation of the root before seed material is generated for the root's DRBG and whenever the root is reseeded.

provides full-entropy output (see Sec. 3.2.2.2). The root may provide prediction resistance if reseeded by the initial random source.

7.1.2.2. Instantiating and Reseeding a Non-Root RBGC Construction

Each non-root RBGC construction in a chain is instantiated by a single RBGC construction (i.e., its parent) using that parent as its randomness source. If the child RBGC construction can be reseeded, the parent normally serves as the randomness source during the reseeding process. However, if the parent is not available for reseeding (e.g., the implementation of the RBGC construction has been moved to a different core on the computing platform), a sibling of the parent may be used as an alternative randomness source provided that:

1. The sibling has been validated for compliance with an RBGC construction, and
2. The DRBG within the sibling supports the security strength of the DRBG to be reseeded.

Using Fig. 29, consider $RBGC_3$ as the target RBGC construction to be reseeded. $RBGC_2$ is the parent of $RBGC_3$ and would normally be used as the randomness source for reseeding $RBGC_3$. If $RBGC_2$ is not available when $RBGC_3$ needs to be reseeded, then a sibling of $RBGC_2$ may be used as an alternative randomness source for reseeding if it meets conditions 1 and 2 above. In Fig. 29, $RBGC_4$ is depicted as a sibling of $RBGC_2$, so $RBGC_4$ may be used as an alternative randomness source (as indicated by the path of black dots) if it is validated for that purpose and the DRBG within the $RBGC_4$ construction can support the security strength of $RBGC_3$'s DRBG.

Implementers of an RBGC tree that use siblings for reseeding the DRBG of an RBGC construction will require a means of recognizing that the parent randomness source is not available and for the parent's sibling(s) to recognize the validity of the request for the generation of seed material and the internal state (within the sibling) to be used for the generation process. Additionally, non-root RBGC constructions cannot guarantee prediction resistance since their randomness sources cannot provide fresh entropy. However, non-root RBGC constructions **should** be reseeded periodically to defend against a potential undetected compromise of the internal state.

7.2. Conceptual Interfaces

An RBGC construction can support instantiation and generation requests (see Sec. 7.2.1 and 7.2.2, respectively) and may provide a capability to be reseeded (see Sec. 7.2.3).

7.2.1. RBGC Instantiation

The DRBG within an RBGC construction may be instantiated by an application at any security strength possible for the DRBG design that does not exceed the security strength of its randomness source. This is accomplished using the **DRBG_Instantiate** function discussed in Sec. 2.8.1.1 and SP 800-90A.

The (target) DRBG in an RBGC construction is instantiated by an application using the following request:

(*status, RBGCx_DRBG_state_handle*) =
DRBG_Instantiate_request(*s, personalization_string*),

where *s* is the requested security strength for the DRBG. The **DRBG_Instantiate_request** received by the DRBG results in the execution of the **DRBG_Instantiate** function in the DRBG with the input in the **DRBG_Instantiate_request** provided as input to the **DRBG_Instantiate** function.

(*status, RBGCx_DRBG_state_handle*) =
DRBG_Instantiate(*s, personalization_string*).

The target DRBG in the RBGC construction cannot be instantiated at a higher security strength than that which is supported by its randomness source. If the target DRBG is successfully instantiated, *RBGCx_DRBG_state_handle* is the state handle returned to the application for subsequent access to the internal state of the DRBG instantiation within the RBGC construction. If the DRBG is implemented to only allow a single internal state, then a state handle is not required. If the instantiation request is invalid (e.g., the requested security strength cannot be provided by the DRBG design or the randomness source; see SP 800-90A), an error indication is returned as the *status* with an invalid state handle.

7.2.1.1. Instantiation of the Root RBGC Construction

The randomness source for the root RBGC construction (also referred to as the initial randomness source) is:

- A validated RBG3(XOR) or RBG3(RS) construction, as specified in Sec. 6;
- A validated RBG2(P) or RBG2(NP) construction, as specified in Sec. 5; or
- A validated full-entropy source that is either:
    - An entropy source that provides output with full entropy, as specified in SP 800-90B, or
    - The output of an SP 800-90B-compliant entropy source that has been externally conditioned by a vetted conditioning function (as specified in Sec. 3.2.2.2) to provide output with full entropy.

When used as the initial randomness source, an RBG3 construction or a full-entropy source can support any valid security strength for the DRBG within the root RBGC construction (i.e., 128, 192, or 256 bits).

When used as the initial randomness source, an RBG2(P) or RBG2(NP) construction can support any security strength for the DRBG within the root RBGC construction that does not exceed the instantiated security strength of the DRBG within the RBG2(P) or RBG2(NP) construction. For example, if the initial randomness source is an RBG2(P) construction whose DRBG is instantiated at a security strength of 128 bits, then the DRBG within the root RBGC construction can only be instantiated at a security strength of 128 bits.

An RBGC designer must consider how to find an available randomness source and how to access it.

7.2.1.1.1. Instantiating the DRBG in the Root Using an RBG2 or RBG3 Construction as the Initial Randomness Source

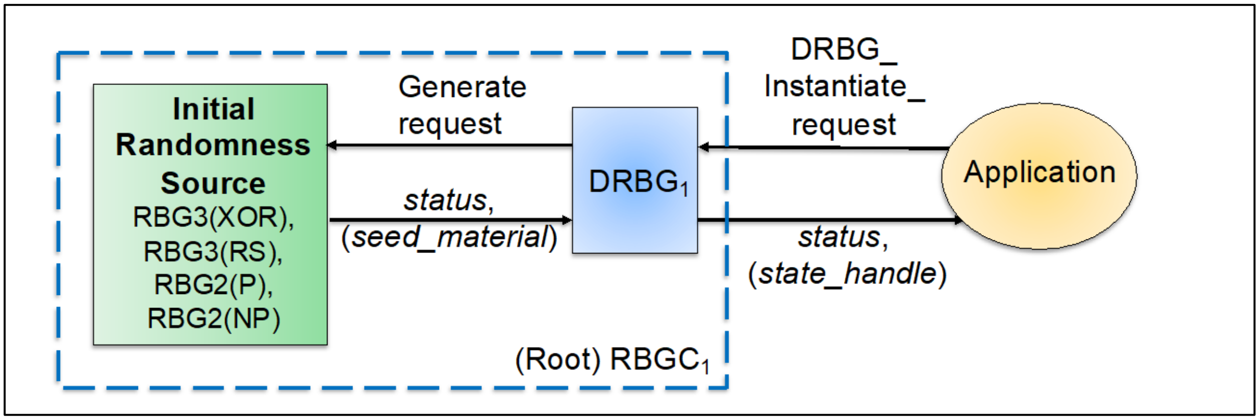

Fig. 30. Instantiation of the DRBG in the root RBGC construction using an RBG2 or RBG3 construction as the randomness source

Figure 30 depicts a request for instantiation of the root RBGC construction by an application. Let $RBGC_1$ be the root and $DRBG_1$ be its DRBG. In this section, the initial randomness source is either an RBG2 or RBG3 construction.

Upon receiving a valid instantiation request from an application (see Sec. 7.2.1), the **DRBG_Instantiate** function within $DRBG_1$ processes the request by obtaining seed material from the initial randomness source. Within the **DRBG_Instantiate** function (in $DRBG_1$), the randomness source is accessed using a **Get_randomness-source_input** call (see SP 800-90A), which is replaced as specified below.

Let s be the intended security strength of $DRBG_1$ in the root RBGC construction.

1. When the DRBG in the root RBGC construction uses a CTR_DRBG <u>without</u> a derivation function, $s + 128$ bits[24] **shall** be obtained from the initial randomness source.
    a. If the randomness source is an RBG2(P) or RBG2(NP) construction, the RBG2 construction **shall** be reseeded before requesting seed material. The **Get_randomness-source_input** call becomes:
        - *status* = **DRBG_Reseed_request**(*RBG2_DRBG_state_handle*, *additional_input*).
        - If (*status* ≠ SUCCESS), then return (*status*, *invalid_state_handle*).

[24] 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.

- (*status, seed_material*) = **DRBG_Generate_request**(*RBG2_DRBG_state_handle, s + 128, s, additional_input*).
- If (*status* ≠ SUCCESS), then return (*status*, *invalid_state_handle*).

RBG2_DRBG_state_handle is the state handle for the internal state of the DRBG within the RBG2 construction. Reseed and generate requests received by an RBG2 construction are discussed in Sec. 5.2.3 and 5.2.2, respectively.

b. If the randomness source is an RBG3(XOR) or RBG3(RS) construction, the **Get_randomness-source_input** call becomes:

- (*status*, *seed_material*) = **RBG3_DRBG_Generate_request**(*s* + 128, *additional_input*).
- If (*status* ≠ SUCCESS), then return (*status*, *invalid_state_handle*).

RBG3_DRBG_state_handle is the state handle for the internal state of the DRBG within the RBG3 construction. An **RBG3_DRBG_Generate_request** received by an RBG3 construction is discussed in Sec. 6.4.1.2 and 6.5.1.2 (the RBG3(XOR) and RBG3(RS) constructions, respectively).

2. For CTR_DRBG (<u>with</u> a derivation function), Hash_DRBG, and HMAC_DRBG, 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) or RBG2(NP) construction, the RBG2 construction **shall** be reseeded before requesting seed material. The **Get_randomness-source_input** call becomes:

- *status* = **DRBG_Reseed**(*RBG2_DRBG_state_handle, additional_input*).
- If (*status* ≠ SUCCESS), then return (*status*, *invalid_state_handle*).
- (*status, seed_material*) = **DRBG_Generate_request**(*RGB2_DRBG_state_handle, 3s/2, s, additional_input*).
- If (*status* ≠ SUCCESS), then return (*status*, *invalid_state_handle*).

RBG2_ DRBG_state_handle is the state handle for the internal state of the DRBG within the RBG2 construction. Reseed and generate requests received by an RBG2 construction are discussed in Sec. 5.2.3 and 5.2.2, respectively.

b. If the randomness source is an RBG3(XOR) or RBG3(RS) construction, the **Get_randomness-source_input** call becomes:

- (*status*, *seed material*) = **RBG3_DRBG_Generate_request**(3*s*/2, *additional_input*).
- If (*status* ≠ SUCCESS), then return (*status*, *invalid_state_handle*).

RBG3_DRBG_state_handle is the state handle for the internal state of the DRBG within the RBG3 construction. An **RBG3_DRBG_Generate_request** received by an RBG3 construction is discussed in Sec. 6.4.1.2 and 6.5.1.2 (the RBG3(XOR) and RBG3(RS) constructions, respectively).

7.2.1.1.2. Instantiating the Root RBGC Construction Using a Full-Entropy Source as the Randomness Source

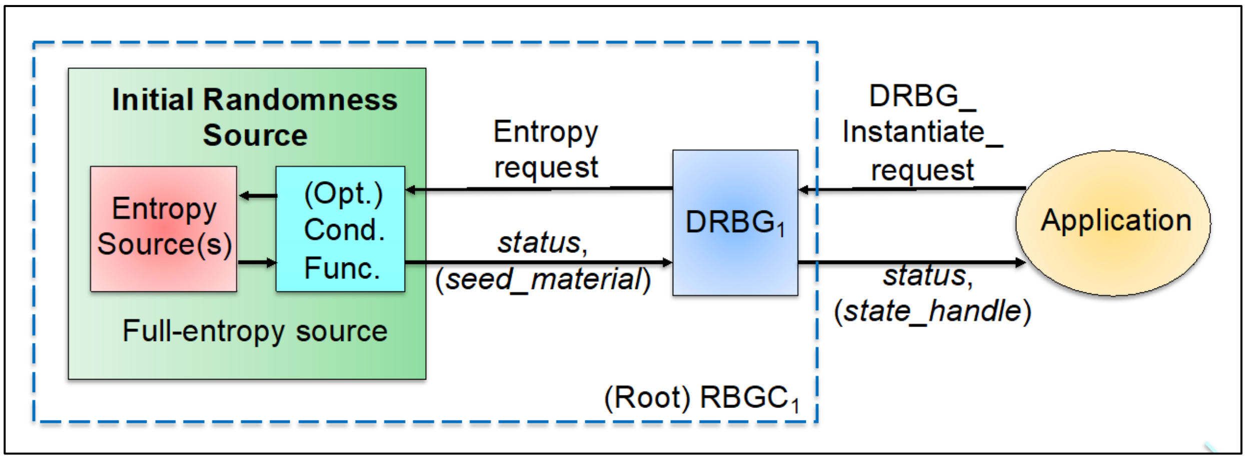

Fig. 31. Instantiation of the DRBG in the root RBGC construction using a full-entropy source as a randomness source

Figure 31 depicts a request for instantiation of the root RBGC construction by an application. Let $RBGC_1$ be the root and $DRBG_1$ be its DRBG. In this section, the initial randomness source is a full-entropy source (see Sec. 7.2.1.1).

Upon receiving a valid instantiation request from an application, the **DRBG_Instantiate** function within $DRBG_1$ continues processing the request by obtaining seed material from the full-entropy source. The full-entropy source may consist of physical or non-physical entropy sources or both, and either Method 1 or Method 2 may be used to count entropy (see Sec. 2.3). Instantiation is performed for an RBG2 construction, as specified in Sec. 5.2.1.

7.2.1.2. Instantiating an RBGC Construction Other Than the Root

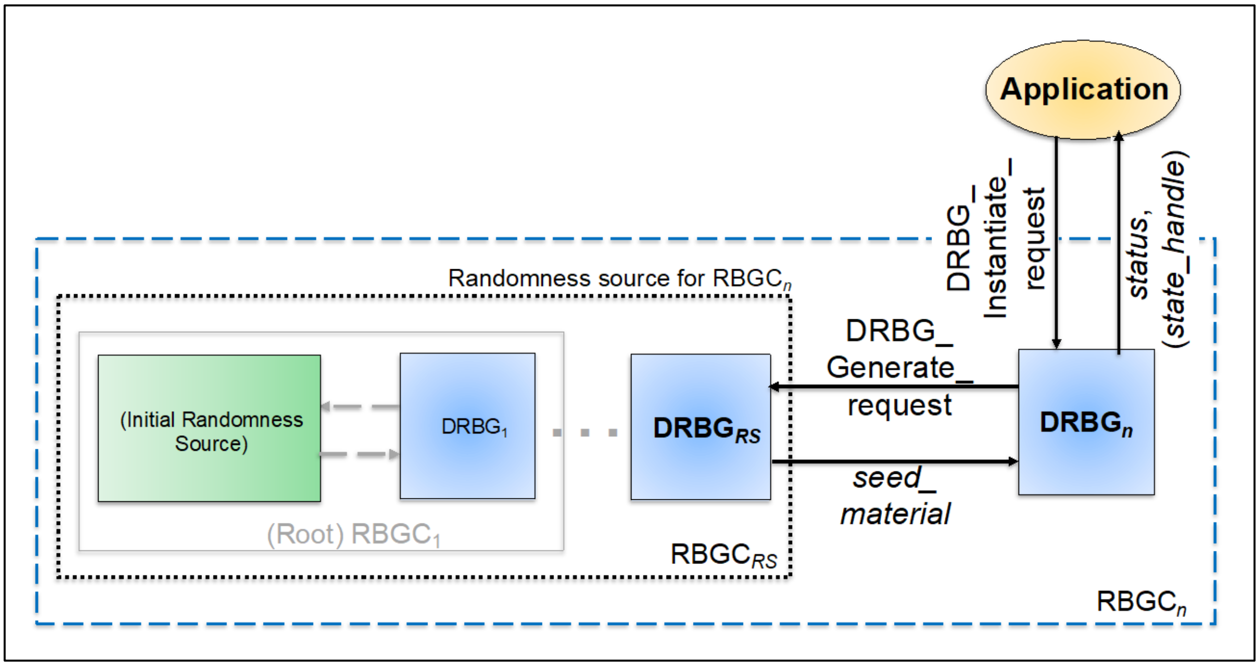

Fig. 32. Instantiation of the DRBG in RBGC$_n$ using RBGC$_{RS}$ as the randomness source

Figure 32 depicts a request by an application for the instantiation of the DRBG within an RBGC construction that is not the root. Let RBGC$_n$ be the RBGC construction receiving the instantiation request, and let DRBG$_n$ be its DRBG. RBGC$_n$ needs to determine the RBGC construction that will serve as its randomness source. The randomness source for a DRBG in an RBGC construction that is not the root of the DRBG chain is the RBGC construction that will immediately precede it in the chain as its parent. Let RBGC$_{RS}$ be the randomness source for RBGC$_n$, and let DRBG$_{RS}$ be its DRBG (see Fig. 32). RBG$_{RS}$ could be the root RBGC construction. RBGC$_1$ is outlined in gray in the figure.

Upon receiving a valid instantiation request from an application, such as

$$(status,\ RBGC\_DRBGn\_state\_handle) = \textbf{DRBG\_Instantiate\_request}(s,\ personalization\_string),$$

DRBG$_n$ executes its **DRBG_Instantiate** function within DRBG$_n$ and processes the request by obtaining seed material from its intended parent randomness source (RBGC$_{RS}$). The **Get_randomness-source_input** call in the **DRBG_Instantiate** function in DRBG$_n$ is replaced as specified below.

Let s be the intended security strength of the DRBG in RBGC$_n$ (shown as DRBG$_n$ in the figure).

1. When $RBGC_n$ is instantiating a CTR_DRBG without a derivation function, $s + 128$ bits[25] **shall** be obtained from the randomness source (i.e., $RBGC_{RS}$) by replacing the **Get_randomness-source_input** call with:

   - (*status, seed_material*) = **DRBG_Generate_request**(*$RBGC_{RS}$_DRBG_state_handle, s + 128, s, additional_input*).
   - If (*status* ≠ SUCCESS), then return (*status*, *invalid_state_handle*).

   $RBGC_{RS}$_DRBG_state_handle is the state handle for the internal state of the DRBG within $RBGC_{RS}$. Upon receiving the **DRBG_Generate_request**, $RBGC_{RS}$ executes its **DRBG_Generate** function (see Sec. 2.8.1.1 and 7.2.2) and checks its output That is,

   - (*status, seed_material*) = **DRBG_Generate**(*$RBGC_{RS}$_DRBG_state_handle, s + 128, s, additional_input*).
   - If (*status* ≠ SUCCESS), then return (*status*, *invalid_state_handle*).

2. For CTR_DRBG (with a derivation function), Hash_DRBG, and HMAC_DRBG, $3s/2$ bits **shall** be obtained from the randomness source ($RBGC_{RS}$) by replacing the **Get_randomness-source_input** call with:

   - (*status, seed_material*) = **DRBG_Generate_request**(*$RBGC_{RS}$_DRBG_state_handle, 3s/2, s, additional_input*).
   - If (*status* ≠ SUCCESS), then return (*status*, *invalid_state_handle*).

   $RBGC_{RS}$_DRBG_state_handle is the state handle for the internal state of the DRBG within $RBGC_{RS}$. Upon receiving the **DRBG_Generate_request**, $RBGC_{RS}$ executes its **DRBG_Generate** function (see Sec. 2.8.1.1 and 7.2.2) and checks its output. That is,

   - (*status, seed_material*) = **DRBG_Generate**(*$RBGC_{RS}$_DRBG_state_handle, 3s/2, s, additional_input*).
   - If (*status* ≠ SUCCESS), then return (*status*, *invalid_state_handle*).

Section 7.2.2 specifies the behavior of the DRBG in an RBGC construction when it receives a generate request. The *status* and any generated *seed_material* are returned to the requesting DRBG ($DRBG_n$) in response to the **DRBG_Generate_request**.

[25] For AES, the block length is 128 bits, and the key length is equal to the security strength s. SP 800-90Ar1 requires the randomness input from the randomness source to be key length + block length bits when a derivation function is not used.

7.2.2. Requesting the Generation of Pseudorandom Bits From an RBGC Construction

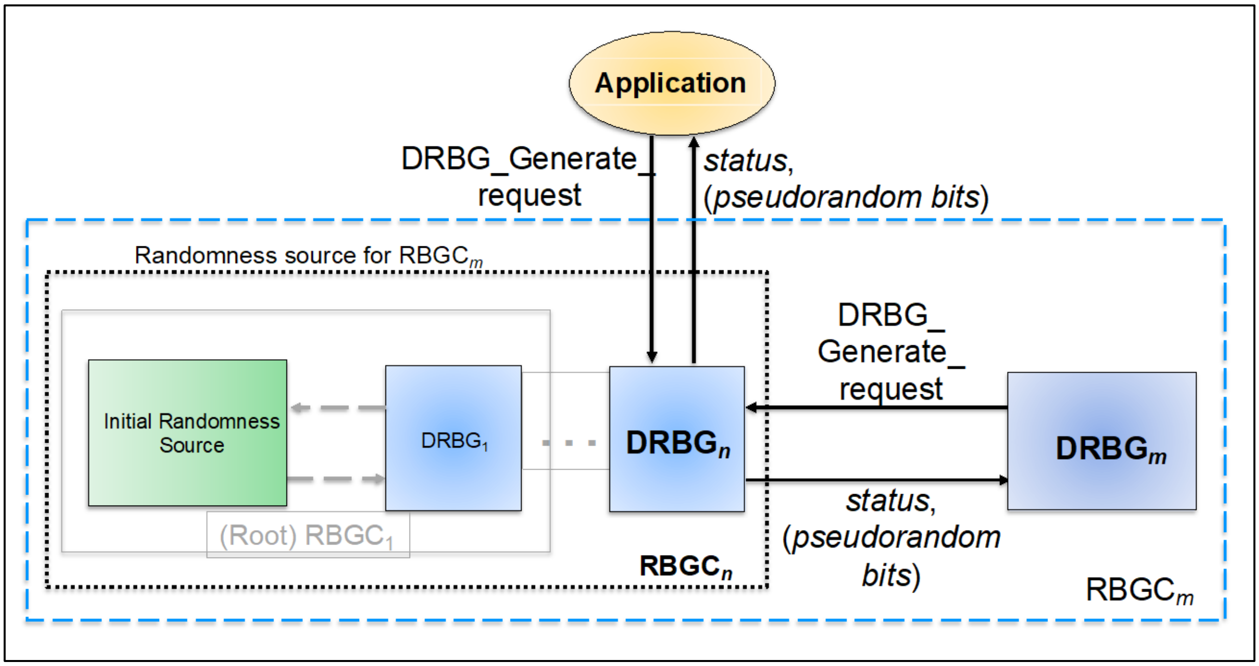

Fig. 33. Generate request received by the DRBG in an RBGC construction

Figure 33 depicts a generate request received by the DRBG in an RBGC construction (i.e., DRBG$_n$ in RBGC$_n$) from a requesting entity (either an application or a DRBG in another RBGC construction, shown as DRBG$_m$ and RBGC$_m$ in the figure). When the requesting entity is DRBG$_m$ (rather than an application), DRBG$_m$ is attempting to be seeded or reseeded with seed material. DRBG$_n$ **shall** be either 1) the parent randomness source for DRBG$_m$ or 2) a sibling of DRBG$_m$'s parent randomness source that meets the requirements of an alternative randomness source (see Sec. 7.1.2.2). RBGC$_n$ could be the root DRBG (the root is outlined in gray in the figure).

The generate request from the requesting entity for this example is:

(*status*, *returned_bits*) = **DRBG_Generate_request**(*RBGCn_DRBG_state_handle*, *requested_number_of_bits*, *requested_security_strength*, *additional_input*),

where *RBGCn_DRBG_state_handle* is the state handle for the internal state of the DRBG in the RBGC construction receiving the generate request (RBGC$_n$). If the **DRBG_Generate_request** received by RBGC$_n$ can be handled, the **DRBG_Generate** function in DRBG$_n$ is executed:

(*status*, *returned_bits*) = **DRBG_Generate**(*RBGCn_DRBG_state_handle*, *requested_number_of_bits*, *requested_security_strength*, *additional_input*).

The **DRBG_Generate** function within DRBG$_n$ processes the generate request.

1. If the generate request cannot be fulfilled (e.g., the requested security strength cannot be provided by the DRBG design used in DRBG$_n$; see SP 800-90A), only an error *status* is returned to the requesting entity. No other output is provided.

2. Otherwise, $DRBG_n$ generates the *requested_number_of_bits* and provides them to the requesting entity in response to the **DRBG_Generate_request** with a *status* of SUCCESS.

7.2.3. Reseeding an RBGC Construction

The reseeding of an RBGC construction is optional. If a reseed capability is implemented within the DRBG of an RBGC construction, the RBGC construction may receive a reseed request from an application, or the DRBG within the construction may reseed itself based on implementation-selected criteria, such as time, number of outputs, events, or — in the case of the root RBGC construction using a full-entropy source — the availability of sufficient entropy.

Section 7.2.3.1 discusses the reseeding of the DRBG in the root RBGC construction. Section 7.2.3.2 discusses the reseeding of the DRBG in an RBGC construction other than the root.

A reseed request from an application is:

(*status*) = **DRBG_Reseed_request**(*RBGCx_DRBG_state_handle, additional_input*),

where *RBGCx_DRBG_state_handle* is the state handle for the internal state of the DRBG in the RBGC construction receiving the reseed request ($RBGC_x$).[26] The **DRBG_Reseed_request** received by $RBGC_x$ results in the execution of $DRBG_x$'s **DRBG_Reseed** function (see Sec. 2.8.1.3). The *status* returned from the **DRBG_Reseed** function **shall** be returned to the application in response to the **DRBG_Reseed_request**.

If the reseed request is invalid (e.g., the state handle is not correct or the DRBG does not have a reseed capability), an error indication is returned as the *status* to the application (i.e., the DRBG has not been reseeded).

Reseeding based on implementation-selected criteria is not initiated by a **DRBG_Reseed_request** from an application but is addressed in Sec. 7.2.3.1 and 7.2.3.2.

7.2.3.1. Reseed of the DRBG in the Root RBGC Construction

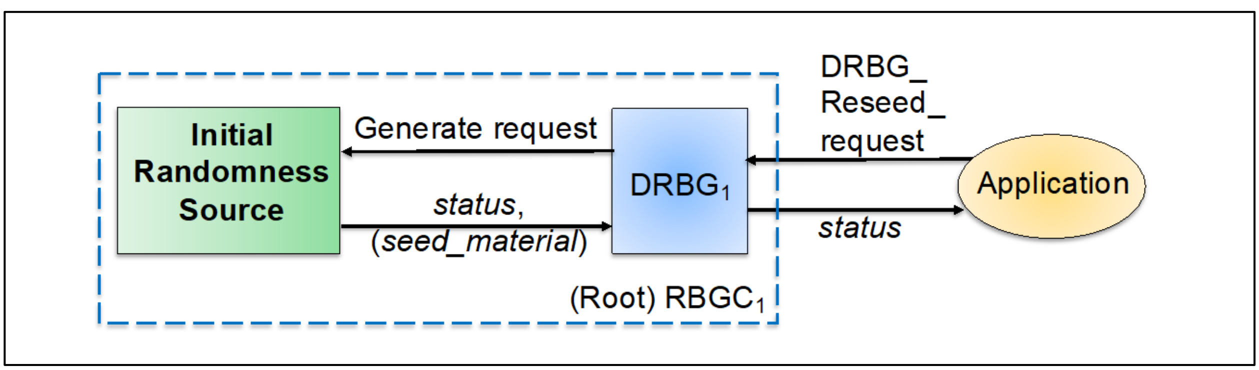

Fig. 34. Reseed request received by the DRBG in the root RBGC construction

[26] For Fig. 34 in Sec. 7.2.3.1, $x = 1$. For Fig. 35 in Sec. 7.2.3.2, $x = n$.

If the root RBGC construction includes a reseed capability (as shown in Fig. 34), the DRBG in the root RBGC construction (e.g., $RBGC_1$) may receive a request from an application for reseeding.

Upon the receipt of a valid reseed request or when reseeding is to be performed based on implementation-selected criteria, the DRBG in the root RBGC construction (e.g., $DRBG_1$) executes its **DRBG_Reseed** function to obtain randomness from the initial randomness source for reseeding itself. This process results in fresh entropy provided by the initial randomness source so that the next output generated by $DRBG_1$ has prediction resistance.

1. When the DRBG in the root RBGC construction uses the CTR_DRBG without a derivation function, reseeding is performed in the same manner as for instantiation.
   - If the initial randomness source is an RBG3(XOR), RBG3(RS), RBG2(P), or RBG2(NP) construction, input is obtained from the initial randomness source as specified in item 1 of Sec. 7.2.1.1.1.
   - If the initial randomness source is a full-entropy source, input is obtained as specified in item 1 of Sec. 7.2.1.1.2.
2. When the DRBG in the root RBGC construction uses the CTR_DRBG (with a derivation function), Hash_DRBG, or HMAC_DRBG, input is obtained from the initial randomness source in the same manner as for instantiation except that s bits are requested (instead of $3s/2$ bits), where s is the instantiated security strength of the DRBG in the root.
   - If the initial randomness source is an RBG3(XOR), RBG3(RS), RBG2(P), or RBG2(NP) construction, input is obtained from the initial randomness source as specified in item 2 of Sec. 7.2.1.1.1.
   - If the initial randomness source is full-entropy source, input is obtained as specified in item 2 of Sec. 7.2.1.1.2.

7.2.3.2. Reseed of the DRBG in an RBGC Construction Other Than the Root

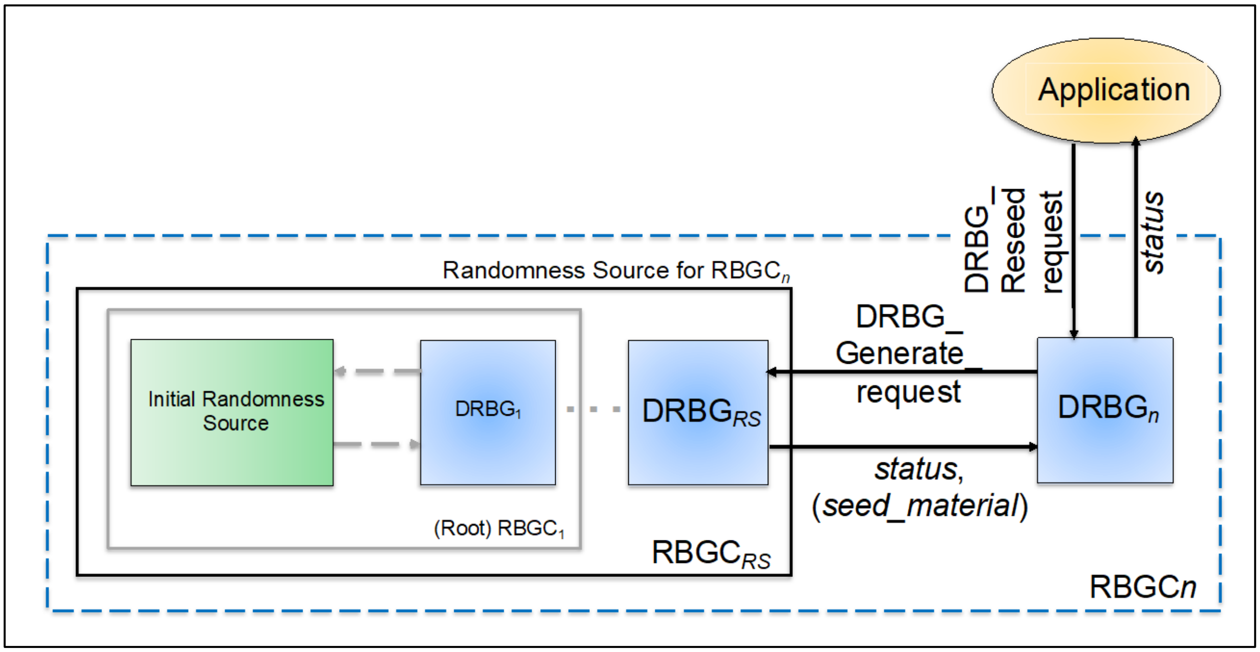

Fig. 35. Reseed request received by an RBGC construction other than the root

As shown in Fig. 35, a DRBG in an RBGC construction other than the root (e.g., RBGC$_n$) may receive a request for reseeding from an application. DRBG$_n$ may also reseed itself based on implementation-selected criteria.

Let DRBG$_{RS}$ be the randomness source to be used for reseeding. DRBG$_{RS}$ **must** be either DRBG$_n$'s parent randomness source or a sibling of the parent (see Sec. 7.1.2.2). DRBG$_{RS}$ may be the DRBG of the root RBGC construction (outlined in gray in the figure). Prediction resistance is not provided for the DRBG being reseeded (DRBG$_n$) since fresh entropy is not provided by the randomness source in this case (DRBG$_{RS}$).

Upon the receipt of a valid reseed request or when a reseed is to be performed based on implementation-selected criteria, the DRBG in RBGC*n* executes its **DRBG_Reseed** function (if implemented). The **Get_randomness-source_input** request in the **DRBG_Reseed** function is replaced by the following:

- (*status, seed_material*) = **DRBG_Generate_request**(*RBGC*$_{RS}$*_DRBG_state_handle, s, s, additional_input*).
- If (*status* ≠ SUCCESS), then return (*status*, *invalid_bitstring*),

where *RBGC*$_{RS}$*_DRBG_state_handle* is the state handle for the internal state of the DRBG in the randomness source (i.e., RBGC$_{RS}$). Upon receiving the request, RBGC$_{RS}$ executes its **DRBG_Generate** function. A *status* indication will be returned from RBGC$_{RS}$ along with seed material if the *status* indicates a success (see Sec. 7.2.2).

Upon the receipt of a response from the randomness source (RBG_{RS}), the DRBG in $RBGC_n$ proceeds as follows:

1. If an error indicator is received from the randomness source ($RBGC_{RS}$) in response to the generate request, the error indicator is forwarded to the application as the *status* in the response to the reseed request.

2. If an error indicator is not received from the randomness source (i.e., $RBGC_{RS}$) and *seed_material* is provided, the *seed_material* is incorporated into the internal state of the DRBG in RBGC*n* as specified in its **DRBG_Reseed** function. If the reseeding of the DRBG in RBGC*n* was in response to a **DRBG_Reseed_request** from an application, the *status* received from the randomness source is returned to the application.

7.3. RBGC Requirements

7.3.1. General RBGC Construction Requirements

An RBGC construction has the following general testable requirements (i.e., testable by the validation labs):

1. An **approved** DRBG from SP 800-90A whose components are capable of providing the targeted security strength for an RBGC construction **shall** be employed.

2. RBGC components **shall** be successfully validated for compliance with SP 800-90A, SP 800-90B, SP 800-90C, FIPS 140, and the specification of any other **approved** algorithm used within the RBGC construction, as applicable.

3. An RBGC construction **shall not** produce any output until it is instantiated.

4. An RBGC construction **shall not** provide output for generating requests that specify a security strength greater than the instantiated security strength of its DRBG.

5. If a health test on the DRBG in an RBGC construction fails, the DRBG instantiation **shall** be terminated.

6. The seed material provided to the DRBG within an RBGC construction **shall** remain secret during transfer from the DRBG's randomness source and remain unobservable from outside its RBG boundary.

7. The internal state of the DRBG within an RBGC construction **shall** remain unobservable from outside its RBG boundary.

8. A tree of RBGC constructions and the initial randomness source for the root RBGC construction **shall** be implemented and operated on a single, physical platform. See Appendix A.3 for further discussion.

9. The initial randomness source **shall not** be removable from the computing platform during operation. If a replacement is required, the root **shall** be instantiated using the replaced randomness source.

10. The seed material **shall not** be output from the computing platform on which it was generated.

11. The internal state of the DRBG within an RBGC construction **shall not** be removed from the computing platform on which it was created, including for storage, and **shall** only be available to the DRBG instantiation for which it was created.

12. If the (parent) randomness source for an RBGC construction is not available for reseeding, the DRBG in the RBGC construction may continue to generate output without reseeding or may be reseeded using a sibling of the parent that has been appropriately validated. When used as an alternative randomness source for reseeding, the sibling **shall** have been validated as an RBGC construction.

General requirements for an RBGC construction that are non-testable are:

13. Each RBGC construction **must** be able to determine the type of randomness source available for its use and how to access it.

14. The randomness source for an RBGC construction **must** provide the requested number of bits at a security strength of s bits or higher, where s is the targeted security strength for that RBGC construction.

15. The specific output of the randomness source (or portion thereof) that is used for the instantiation or reseed of an RBGC construction **must not** be used for any other purpose, including for seeding or reseeding a different instantiation or RBGC construction.

16. The output of an RBGC construction **must not** be used as seed material for a predecessor (e.g., ancestor) RBGC construction.

7.3.2. Additional Requirements for the Root RBGC Construction

An RBGC construction that is used as the root of a DRBG chain has the following additional testable requirements (i.e., testable by the validation labs):

1. For CTR_DRBG (with a derivation function), Hash_DRBG, or HMAC_DRBG, $3s/2$ bits **shall** be obtained from the initial randomness source for instantiation, where s is the targeted security strength for the DRBG used in the RBGC construction. When reseeding, s bits **shall** be obtained from the initial randomness source.

2. For a CTR_DRBG without a derivation function used as the DRBG within the root RBGC construction, $s + 128$ bits[27] **shall** be obtained from the randomness source for instantiation and reseeding, where s is the targeted security strength for the DRBG used in the RBGC construction.

[27] Note that $s + 128 = keylen + blocklen = seedlen$, as specified in SP 800-90Ar1.

3. If the randomness source for the root RBGC construction is an RBG2 construction, a request for reseeding the DRBG in the RBG2 construction **shall** precede a request for generating seed material.

The non-testable requirements for the root RBGC construction are:

4. The initial randomness source for the root RBGC construction **must** be a validated RBG3(XOR), RBG3(RS), RBG2(P), or RBG2(NP) construction or a full-entropy source.
5. A full-entropy source serving as the initial randomness source **must** be either an entropy source that has been validated as providing full-entropy output or a validated entropy source that uses the external conditioning function specified in Sec. 3.2.2.2.
6. The DRBG in the root RBGC construction may be instantiated at any security strength for the design, subject to the following restriction: if the initial randomness source is an RBG2(P) or RBG2(NP) construction, the root **must not** be instantiated at a security strength greater than the security strength of the RBG2(P) or RBG2(NP) construction.

7.3.3. Additional Requirements for an RBGC Construction That is NOT the Root of a DRBG Chain

An RBGC construction that is NOT the root of a DRBG chain has no additional testable requirements beyond those in Sec. 7.3.1.

The non-testable requirements for an RBGC construction that is not the root of a DRBG chain are:

1. Each RBGC construction **must** have only one parent RBGC construction as a randomness source for instantiation and reseeding, although under certain conditions, a sibling of the parent may be used as a randomness source for reseeding (see requirement 12 in Sec. 7.3.1).
2. An RBGC construction **must** reside on the same computing platform as its parent and any alternative randomness source.
3. Each RBGC construction may be instantiated at any security strength for the design that does not exceed the security strength of its parent randomness source.

8. 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 Sec. 8.1). Section 8.2 provides requirements for implementation validation.

8.1. Health Testing

Health testing is the testing of an implementation prior to and during normal operations to determine whether 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 SP 800-90A and SP 800-90B as well as perform health tests on the components of SP 800-90C. FIPS 140 specifies the testing to be performed within a cryptographic module.

8.1.1. 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).

8.1.2. Handling Failures

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

SP 800-90A and SP 800-90B discuss error handling for DRBGs and entropy sources, respectively.

8.1.2.1. Entropy-Source Failures

A failure of a validated entropy source is reported to the **Get_entropy_bitstring** process in response to entropy requests to the entropy source(s). The **Get_entropy_bitstring** function notifies the consuming application of such failures as soon as possible (see item 4 of Sec. 3.1). The consuming application may choose to terminate the RBG operation. Otherwise, the RBG may continue operation if any entropy source credited for providing entropy[28] is still healthy (i.e., a failure has not been reported by those entropy sources).

If all entropy sources credited with providing entropy report failures, the RBG operation **shall** be terminated (e.g., stopped) until such time as the entropy source is repaired and successfully tested for correct operation.

[28] Only the entropy provided by physical entropy sources is credited for the RBG2(P) and RBG3 constructions. Entropy from both physical and non-physical entropy sources is credited for the RBG2(NP) construction. See Sec. 5 and 6.

8.1.2.2. 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 known-answer health tests within the RBG. Failures could also be detected by the system in or on which the RBG resides.

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

8.2. 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 SP 800-90A and SP 800-90B
- 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 has been provided, as specified in SP 800-90C

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 SP 800-90A and SP 800-90B. 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.
- 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 employed.
- 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 Sec. 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 in a security policy).
- If sub-DRBGs can be used in an RBG1 construction, the maximum number of sub-DRBGs that can be supported by the implementation and the security strengths to be supported by the sub-DRBGs.
- For RBG2 and RBG3 constructions, a statement that identifies the conditions under which the DRBG is reseeded (e.g., when requested by a consuming application, at a given time interval, etc.).
- For an RBG3 construction, a statement that indicates whether the DRBG can be accessed directly (i.e., the DRBG internal state used by the RBG3 construction can be accessed using calls directly to the DRBG).
- For an RBG3 construction, the security policy **shall** indicate the fallback security strength that can be supported by the DRBG if the entropy source fails (i.e., the fallback security strength is the instantiated security strength of the DRBG).
- For an RBG3(RS) construction, when implementing CTR_DRBG (with a derivation function), Hash_DRBG, or HMAC_DRBG, the method used for obtaining $s + 64$ bits of entropy to produce s full-entropy bits (see Sec. 6.5.1.2.1)
- For an RBGC construction, whether it is capable of serving as the root of a DRBG chain, how it “finds” an appropriate randomness source for seeding and reseeding (if implemented), whether it can instantiate child RBGC constructions, any restrictions on the number of child RBGC constructions in the implementation, whether it can be used as an alternative randomness source for another RBGC construction and how this is accomplished (see the note in Sec. 7.1.2.2), and whether it can be reseeded.
- If an RBGC construction can serve as the root of a DRBG chain, identify the initial randomness source types that can be used. If the randomness source can be a full-entropy source, describe the entropy sources to be used.
- Documentation specifying the guidance to users about fulfilling the non-testable requirements, as appropriate (see Sec. 4.4, 5.3, 6.3, and 7.3).

References

[AdversarialInput] Hoang, V.T., Shen, Y. (2020) Security Analysis of NIST CTR-DRBG. In: Micciancio, D., Ristenpart, T. (eds) Advances in Cryptology – CRYPTO 2020. CRYPTO 2020. Lecture Notes in Computer Science(), vol 12170. Springer, Cham. https://doi.org/10.1007/978-3-030-56784-2_8

[AIS20] AIS 20: Funktionalitätsklassen und Evaluationsmethodologie für deterministische Zufallszahlengeneratoren (Version 3) (Bundesamt für Sicherheit in der Informationstechnik (BSI)) (2013), Report. Available at https://www.bsi.bund.de/dok/66138618284.

[AIS31] AIS 31: Funktionalitätsklassen und Evaluationsmethodologie für physikalische Zufallszahlengeneratoren (Version 3) (Bundesamt für Sicherheit in der Informationstechnik (BSI)) (2013), Report. Available at https://www.bsi.bund.de/dok/6618252.

[BSIFunc] Peter M, Schindler W (2022) A Proposal for Functionality Classes for Random Number Generators (Version 2.35, DRAFT) (Bundesamt für Sicherheit in der Informationstechnik (BSI)), Report. Available at https://www.bsi.bund.de/dok/ais-20-31-appx-2022.

[FIPS_140] National Institute of Standards and Technology (2001) *Security Requirements for Cryptographic Modul*es. (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

[FIPS_140IG] National Institute of Standards and Technology, Canadian Centre for Cyber Securit*y Implementation Guidance for FIPS 140-2 and the Cryptographic Module Validation Progr*am, [Amended]. Available at https://csrc.nist.gov/csrc/media/projects/cryptographic-module-validation-program/documents/fips140-2/FIPS1402IG.pdf

[FIPS_180] 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

[FIPS_197] National Institute of Standards and Technology (2001) *Advanced Encryption Standard (AE*S). (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 197. https://doi.org/10.6028/NIST.FIPS.197

[FIPS_198] National Institute of Standards and Technology (2008) *The Keyed-Hash Message Authentication Code (HMA*C). (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 198-1. https://doi.org/10.6028/NIST.FIPS.198-1.

[FIPS_202] National Institute of Standards and Technology (2015) *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functio*ns. (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 202. https://doi.org/10.6028/NIST.FIPS.202

[InputLengths] Thomas Shrimpton, R. Seth Terashima: A Provable-Security Analysis of Intel's Secure Key RNG. EUROCRYPT (1) 2015: 77-100. Available at https://eprint.iacr.org/2014/504.pdf

[ISO_18031] ISO Central Secretary (2011) ISO/IEC 18031:2011 Information technology — Security techniques — Random bit generation (International Organization for Standardization, Geneva, CH), Standard ISO/IEC 18031:2011. Available at https://www.iso.org/standard/54945.html.

[NISTIR_8427] 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 (NIST IR) 8427. https://doi.org/10.6028/NIST.IR.8427

[SP_800-22] Rukhin A, Soto J, Nechvatal J, Smid M, Barker E, Leigh S, Levenson M, Vangel M, Banks D, Heckert N, Dray J, Vo S, Bassham L (2010) SP 800-22 Rev. 1a A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications (National Institute of Standards and Technology), Available at https://doi.org/10.6028/NIST.SP.800-22r1a

[SP_800-38B] Dworkin MJ (2005) *Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authenticati*on. (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

[SP_800-57Part1] 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

[SP_800-90A] 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

[SP_800-90B] 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

[SP_800-131A] Barker EB, Roginsky AL (2019) *Transitioning the Use of Cryptographic Algorithms and Key Leng*ths. (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. Auxiliary Discussions (Informative)

A.1. Entropy vs. Security Strength

This appendix compares and contrasts the concepts of *entropy* and *security strength*.

A.1.1. 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.1.2. Security Strength

A deterministic cryptographic mechanism (e.g., the DRBGs defined in SP 800-90A) 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.1.3. 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 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 currently showing 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 and is computationally bounded.

A.1.4. Entropy and Security Strength in This Recommendation

The DRBG within the RBG1 construction is instantiated from either an RBG2(P) or an RBG3 construction. 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. An RBG3 construction supports any desired security strength.

The DRBG within an RBG2 or RBG3 construction is instantiated using a bitstring with a certain amount of entropy obtained from a validated entropy source.[29] 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, instantiating and reseeding a CTR_DRBG without a derivation function requires a bitstring with exactly $s + 128$ full-entropy bits. This bitstring can either be obtained directly from an entropy source that provides full-entropy output or from an entropy source via an **approved** (i.e., vetted) conditioning function (see Sec. 3.2).

RBG3 constructions are designed to provide full-entropy outputs but with a DRBG included in the design as a second security anchor in case the entropy source fails undetectably. Entropy bits are obtained either directly from an entropy source or from an entropy source via an **approved** (i.e., 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 the highest security strength that can be provided by its design (*highest_strength*). If an entropy-source has an undetectable failure, the RBG3 construction outputs are generated at that security strength. In this case, the security strength of a bitstring produced by the RBG is the minimum of *highest_strength* and the length of the bitstring — that is, $security\_strength = \mathbf{min}(highest\_strength, length)$.

The DRBG within an RBGC construction is instantiated using a bitstring from a randomness source. The randomness source for an RBGC construction will be either an initial randomness source (when the RBGC construction is the root of a tree of such constructions) or another RBGC

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

construction. The tree of RBGC constructions will always originate from an **approved** initial randomness source that is either a full-entropy source or an RBG2 or RBG3 construction, each of which includes a validated entropy source.

In conclusion, entropy sources and properly functioning RBG3 constructions provide output with entropy. RBG1, RBG2, and RBGC 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 (i.e., an RBG(P) construction) to produce output at the highest security strength for the DRBG design.

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.

A.2. Generating Full-Entropy Output Using the RBG3(RS) Construction

Table 4 provides information on generating full-entropy output using the RBG3(RS) construction with the DRBGs in SP 800-90A.

Table 4. Values for generating full-entropy bits by an RBG3(RS) construction

DRBG	DRBG Primitives	Highest Security Strength (s) that may be supported by the DRBG	Entropy obtained during a normal reseed operation (r)	Entropy required for s bits with full entropy ($s + 64$)
CTR_DRBG (with no derivation function)	AES-128	128	256	192
	AES-192	192	320	256
	AES-256	256	384	320
CTR_DRBG (using a derivation function)	AES-128	128	128	192
	AES-192	192	192	256
	AES-256	256	256	320
	SHA-256 SHA3-256	256	256	320
	SHA-384 SHA3-384	256	256	320
	SHA-512 SHA3-512	256	256	320

Each DRBG is based on the use of an **approved** hash function or block cipher algorithm as a cryptographic primitive.

- Column 1 lists the DRBG types.
- Column 2 identifies the cryptographic primitives that can be used by the DRBG(s) in column 1.

- Column 3 indicates the highest security strength (s) that can be supported by the cryptographic primitive in column 2.[30]
- Column 4 indicates the amount of fresh entropy (r) that is obtained by a **DRBG_Reseed** function for the security strength identified in column 3, as specified in SP 800-90A.
- Column 5 indicates the amount of entropy required to be inserted into the cryptographic primitive ($s + 64$) to produce s bits with full entropy.

For the CTR_DRBG with no derivation function, the amount of entropy obtained during a reseed as specified in SP 800-90A (see column 4) exceeds the amount of entropy needed to subsequently generate s bits of output with full entropy (see column 5), where s is 128, 192, or 256. Therefore, reseeding as specified in SP 800-90A is appropriate.

However, for the CTR_DRBG that uses a derivation function or the Hash_DRBG or HMAC_DRBG, a reseed as specified in SP 800-90A does not provide sufficient entropy for producing s bits of full-entropy output for each execution of the **DRBG_Generate** function (see columns 4 and 5). Section 6.5.1.2.1 provides two methods for obtaining the required $s + 64$ bits of entropy needed to generate s bits of full-entropy output:

1. Modify the **DRBG_Reseed** function to obtain $s + 64$ bits of entropy from the entropy source(s) rather than the s bits of entropy specified in SP 800-90A. This approach may be used in implementations that have access to the internals of the DRBG implementation.
2. Obtain 64 bits of entropy directly from the entropy source(s) and provide it as additional input when invoking the **DRBG_Reseed** function. As specified in SP 800-90A, the **DRBG_Reseed** function obtains s bits of entropy from the entropy source(s) and concatenates the additional input to it before updating the internal state with the concatenated result (see the specification for the reseed algorithm for each DRBG type in SP 800-90A), thus incorporating $s + 64$ bits of fresh entropy into the DRBG's internal state.

A.3. Additional Considerations for RBGC Constructions

The boundaries for an RBGC construction are more difficult to define than other constructions specified in this document, which makes validation more difficult. This difficulty arises from changes in the structure of the RBGC tree (e.g., RBGC constructions created in software at runtime) and the possibility that the module containing the DRBG of the RBGC construction may be validated separately from the module containing the randomness source that seeds and reseeds it.

This section contains examples of acceptable RBGC constructions as well as designs that properly transmit seed material. To simplify the discussion, the figures show only the DRBG in each RBGC construction. For example, $DRBG_1$ is the DRBG for the $RBGC_1$, which is used in the examples as the root of the tree (i.e., the root DRBG), and $DRBG_2$ is the DRBG for $RBGC_2$.

[30] Columns 2 and 3 provide the same information as **Table 3**.

A.3.1. RBGC Tree Composition

When parts of an RBGC tree are validated separately, the tree can later be composed in a safe manner to ensure that the requirements given in Sec. 7 are met. An RBGC tree consists of an initial randomness source and a root RBGC construction (at a minimum) and may include descendent RBGC constructions (e.g., children and grandchildren). Additional RBGC constructions (called subtrees) may be added to form a more complex tree. Each subtree consists of at least one RBGC construction that may have its own descendants but is unable to access the initial randomness source.

Consider two modules — A and B — that are evaluated separately (see Fig. 36). Module B does not contain a root DRBG, but module A does. Module A contains an initial randomness source and a DRBG that can access the initial randomness source to serve as the root of a tree (shown as $DRBG_1$). Module B does not include an initial randomness source, so no DRBG in that module can serve as a root. The following examples show how DRBGs in module B can be evaluated as RBGC constructions.

The simplest case for tree composition occurs when one RBGC construction satisfies the requirements for the root RBGC, and every other RBGC construction involved meets the requirements of a non-root RBGC construction. Figures 36 and 37 show compositions where module A has been validated as an RBGC tree containing an initial randomness source, a root (shown as $DRBG_1$), two children of the root ($DRBG_2$ and $DRBG_4$), and $DRBG_3$ (a child of $DRBG_2$). Module B contains a subtree consisting of $DRBG_5$ and two child DRBGs ($DRBG_6$ and $DRBG_7$). In these examples, all DRBGs meet the requirements for RBGC constructions.

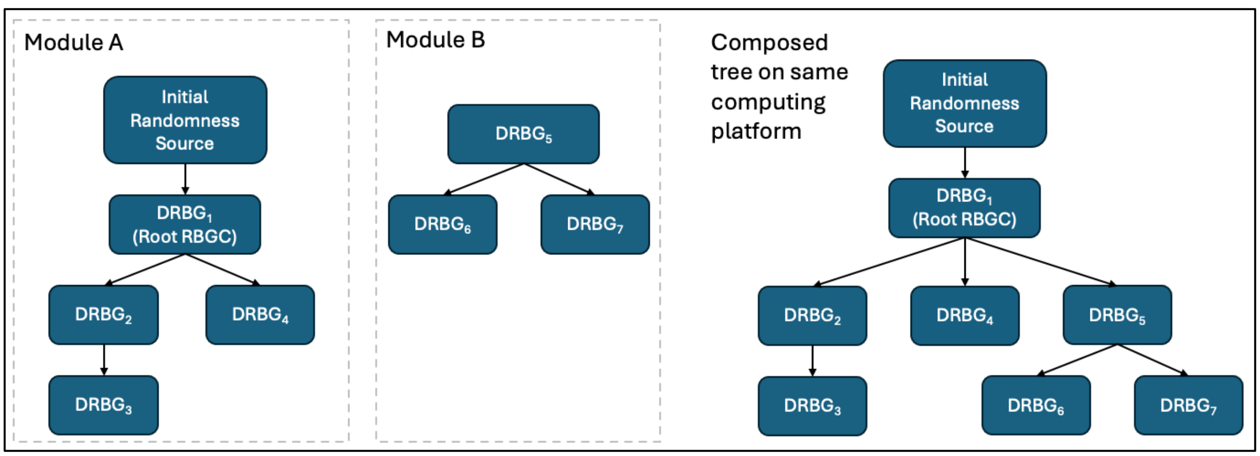

Fig. 36. Subtree in module B seeded by root RBGC of module A

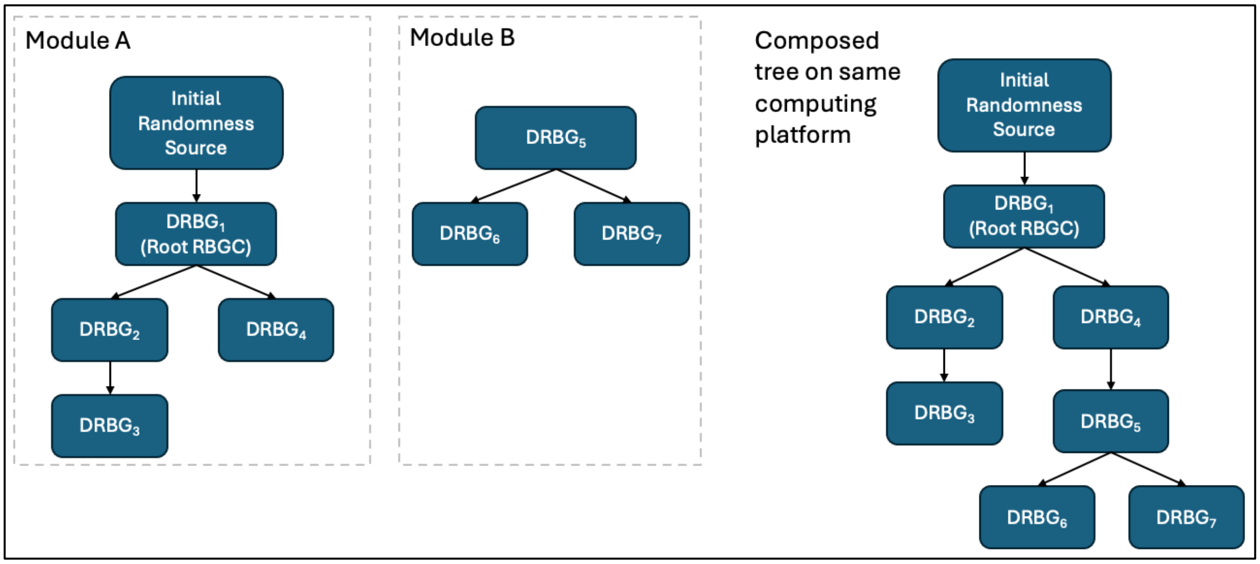

Fig. 37. Subtree in module B seeded by a non-root DRBG of module A (i.e., DRBG$_4$)

In Fig. 36, the DRBGs in module B are added to the tree by using the root (DRBG$_1$) as the randomness source for DRBG$_5$. In Fig. 37, the DRBGs in module B are added to the tree by using DRBG$_4$ as the randomness source for DRBG$_5$.

It is possible to compose trees where some of the DRBGs in module A do not meet the requirements of an RBGC-compliant tree. Figure 38 depicts two DRBGs — DRBG$_2$ and DRBG$_3$ — that do not meet RBGC requirements because a loop exists when DRBG$_3$ is used to reseed DRBG$_2$. The DRBGs in purple boxes connected to the parent through dashed lines do not meet the DRBG requirements for an RBGC construction.

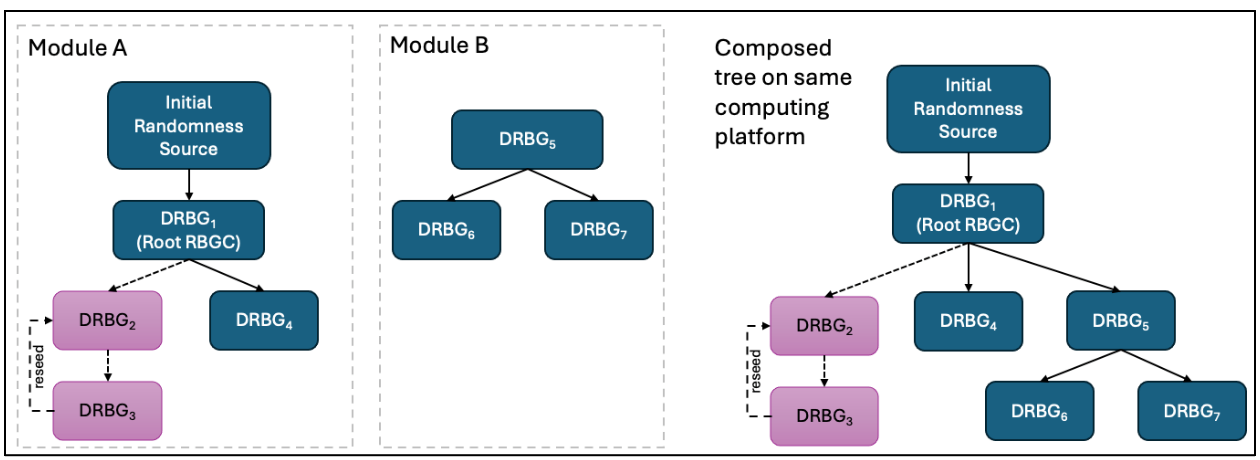

Fig. 38. Subtree in module B seeded by DRBG$_4$ in module A

If module B is added to the tree such that DRBG$_4$ is the randomness source for DRBG$_5$, the elements of module B's subtree only depend on DRBGs that meet RBGC requirements (i.e., DRBG$_1$ and DRBG4) and may therefore be validated as RBGC constructions when added to the tree in this manner.

However, if the DRBGs in module B are added to the tree so that $DRBG_2$ is the randomness source for $DRBG_5$ (see Fig. 39), then the resulting tree is not a compliant RBGC tree.

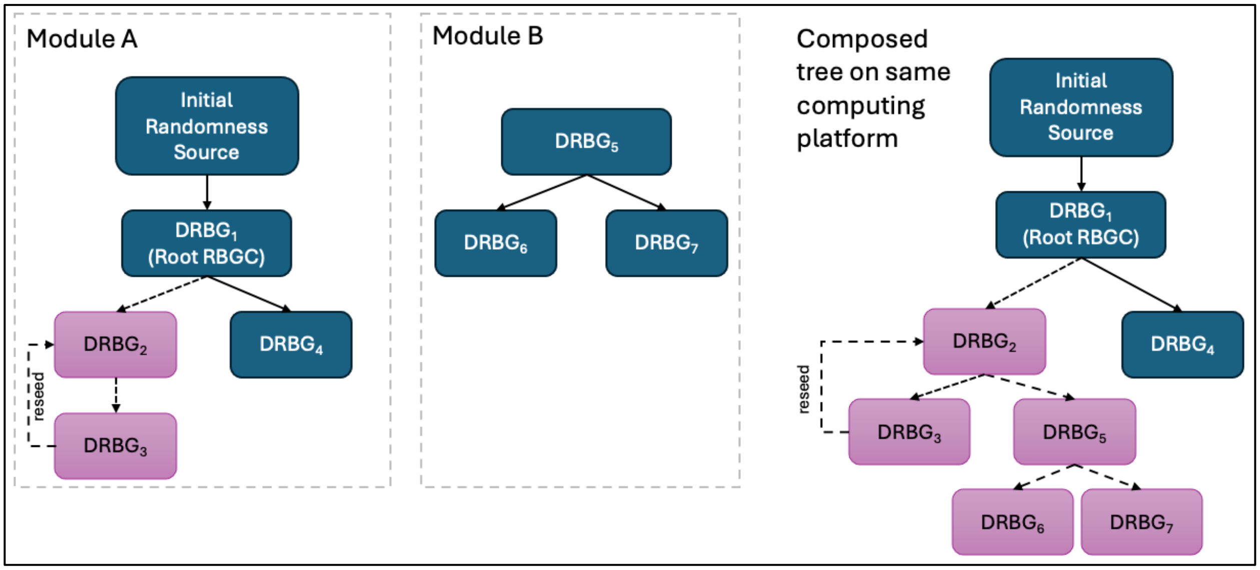

Fig. 39. Subtree in module B seeded by $DRBG_2$ of module A

A.3.2. Changes in the Tree Structure

New RBGC subtrees may be added to the tree during operation, and others may be removed. An RBGC construction may not be moved from one physical platform to another by any means, including backups, snapshots, and cloning.

An RBGC construction could be copied via forking within a single computer platform. Such cases are permissible as long as the original and/or new processes are reseeded prior to fulfilling any requests. This ensures that multiple instances of the same RBGC construction are not operating simultaneously with the same internal states. Without this reseeding, the outputs of one RBGC construction could be used to learn subsequent outputs from its counterpart, voiding any claims of prediction resistance.

A.3.3. Using Virtual Machines

The phrase "same computing platform" (used in Sec. 7) is intended to restrict realizations of RBGC constructions to similar concepts of a randomness source and DRBGs that exist within the same RBG boundary. In particular, seed material must pass from a randomness source to a DRBG in a way that provides the same guarantees as using a physical secure channel.

RBGC constructions used within virtual machines (VMs) pose a unique challenge because they can be on the same physical platform yet communicate through a local area network (LAN). Whether network traffic between VMs is routed solely by the hypervisor's virtual LAN (VLAN) or is sent to the platform's network for routing depends on the configuration of the VLAN. For example, two VMs that are in different port groups or use different virtual switches may transmit the data outside of the physical system they reside on, as shown in Fig. 40 and 41.

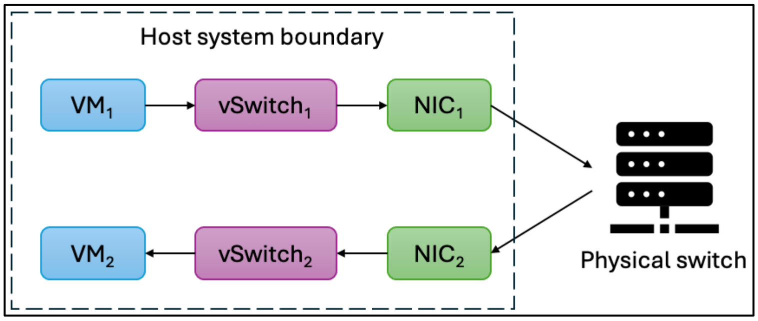

Fig. 40. VM$_1$ and VM$_2$ with different virtual switches

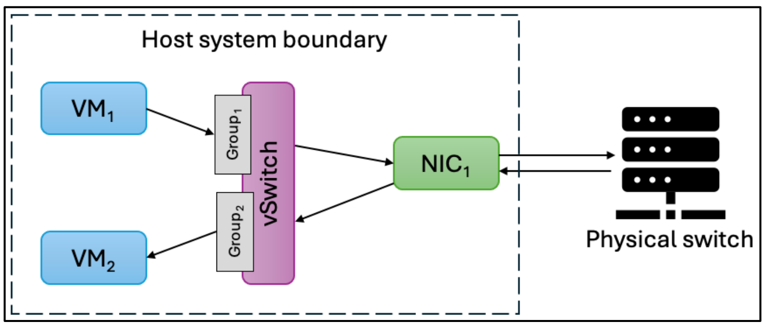

Fig. 41. VM$_1$ and VM$_2$ with the same virtual switch but different port groups

A DRBG within a virtual machine could potentially obtain seed material from sources outside of the virtual machine if the seed material originates on the same computing platform. In particular, seed material can be obtained from randomness sources that reside in levels below the virtual machine, such as a hypervisor, host operating system, or the platform hardware. Figure 42 shows an example in which all seed material is obtained from lower levels on the same system.

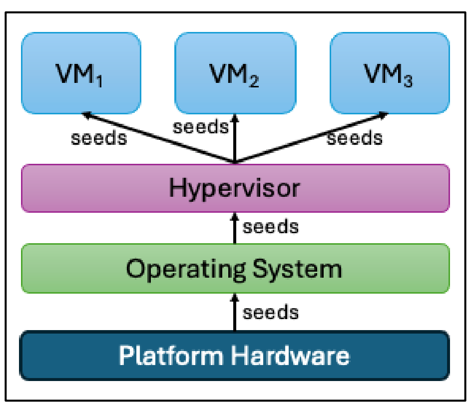

Fig. 42. Acceptable external seeding for virtual machine RBGC constructions

To comply with an RBGC tree as specified in SP 800-90C, virtual machines cannot provide seed material to each other via a virtual network (see Fig. 43).

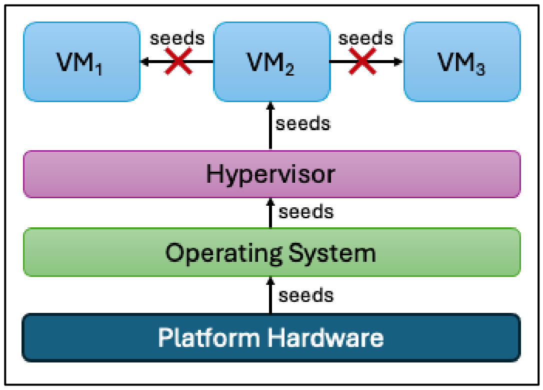

Fig. 43. Acceptable external seeding for an RBGC construction in VM_2 but not in VM_1 and VM_3

This is a very important point in terms of local security guarantees. Virtual network configurations may change without being visible to a VM and alter the path of virtual network traffic. Therefore, it cannot be guaranteed that the seed material will never cross the physical network. Two configuration examples where data transmitted between virtual machines exits the host machine are shown in Fig. 40 and 41.

A.3.4. Reseeding From Siblings of the Parent

There may be situations in which it is acceptable for an RBGC construction to obtain reseeding material from an RBGC construction other than its parent. Figure 44 presents an example of a computing platform with an OS-level RBGC construction and tree containing an initial randomness source, root RBGC construction (containing $DRBG_1$), and three child RBGC constructions, each associated with a different processor (shown as CPU_1, CPU_2, and CPU_3).

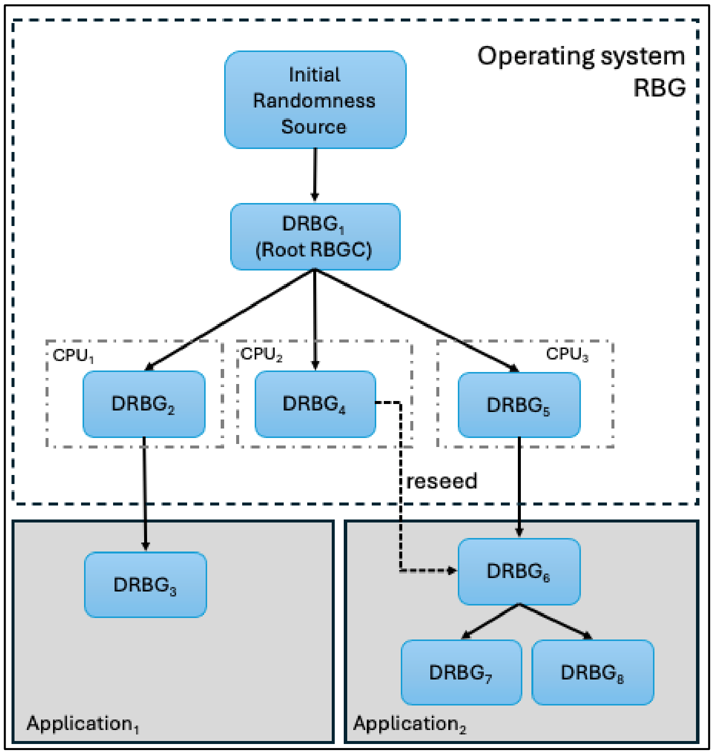

Fig. 44. Application subtree obtaining reseed material from a sibling of its parent

The DRBGs associated with these CPUs are $DRBG_2$, $DRBG_3$, and $DRBG_4$, each of which can be used as a randomness source by application-level RBGC constructions. $Application_2$ contains a subtree of RBGC constructions with $DRBG_6$, $DRBG_7$, and $DRBG_8$. This subtree is composed of the OS-level RBGC at $DRBG_5$ (i.e., $DRBG_5$ is the parent of $DRBG_6$).

Ideally, $DRBG_6$ would obtain bits for reseeding from its parent, $DRBG_5$, but there may be reasons why this is either undesirable (e.g., because of load balancing) or not allowed by the RBGC requirements (e.g., seed material would exit the computing platform). Figure 44 provides an example in which a computing platform is a multi-processor system that performs load balancing to distribute tasks across processors. Application 2 (containing $DRBG_6$) was originally located on CPU_3 so that $DRBG_6$ was originally seeded by $DRBG_5$ (i.e., DRBG5 is the parent of $DRBG_6$). If Application 2 is later moved to CPU_2 and $DRBG_6$ needs to be reseeded, it may be costly to reseed using $DRBG_5$. For efficiency within the multi-processor system, $DRBG_6$ can instead be reseeded using $DRBG_4$ if $DRBG_4$ has been designed and validated to meet the RBGC requirements. Note that $DRBG_4$ and $DRBG_5$ are siblings since they have the same parent ($DRBG_1$).

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.7 provide examples of each RBG construction.

The figures do not show that if an error indicates an RBG failure (e.g., a noise source in the entropy source has failed), the RBG operation is terminated (see Sec. 2.6 and 8.1.2.1). For the examples below, all entropy sources are considered to be physical entropy sources. In order to simplify the examples, the *additional_input* parameter in the generate and reseed requests and generate functions is not used.

B.1. Direct DRBG Access in an RBG3 Construction

An implementation of an RBG3 construction may be designed so that the DRBG implementation used within the construction can be directly accessed by a consuming application using the same or separate instantiations from the instantiation used by the RBG3 construction (see the examples in Fig. 45).

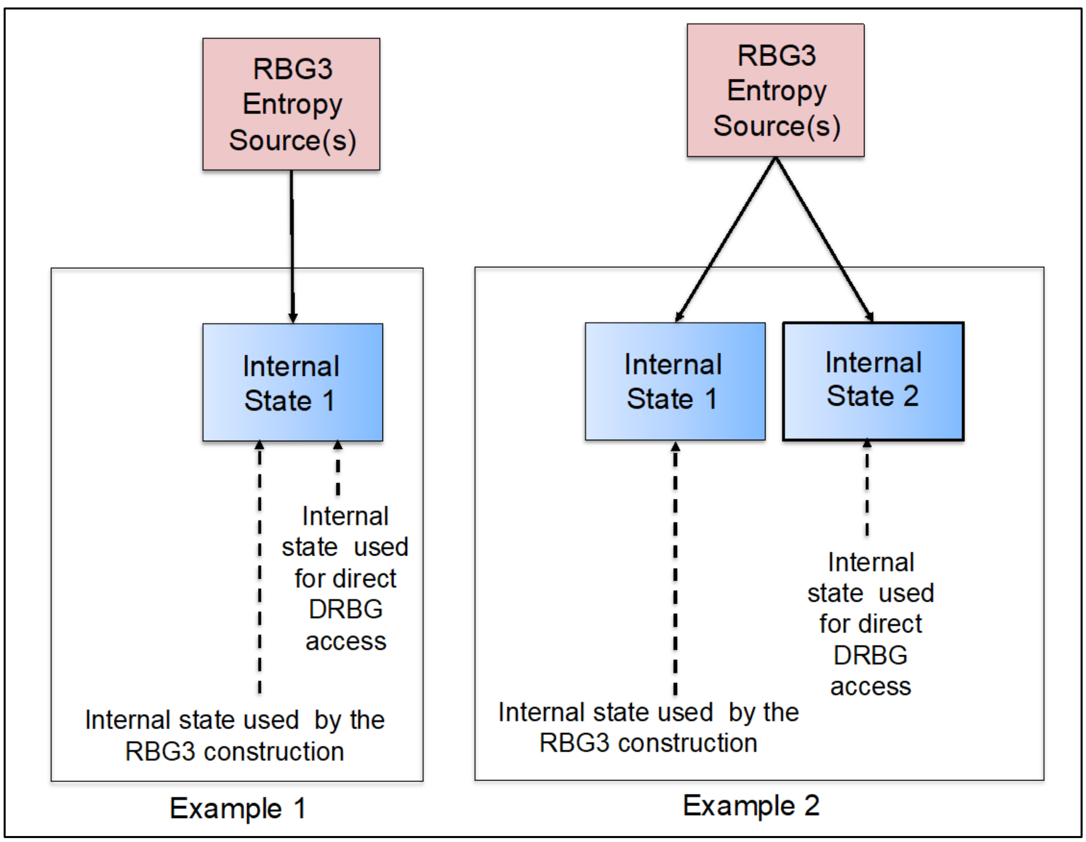

Fig. 45. DRBG Instantiations

In the leftmost example in Fig. 45, 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_DRBG_state handle*). Generation and reseeding for RBG3 operations use RBG3 function calls (see Sec. 6.4 and 6.5), while generation and reseeding for direct DRBG access use RBG2 function calls (see Sec. 5.2) with the *RBG3_DRBG_state_handle*.

In the rightmost example in Fig. 45, the RBG3 construction and directly accessible DRBG use different internal states. 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 *RBG2_DRBG_state_handle*). Generation and reseeding for RBG3 operations use RBG3 function calls and *RBG3_DRBG_state_handle* (see Sec. 6.4 and 6.5), while generation and reseeding for direct DRBG access use RBG2 function calls and *RBG2_DRBG_state_handle* (see Sec. 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 using the same internal state as the RBG3 construction (rather than operating as part of the RBG3 construction), the DRBG operates as an RBG2(P) construction. A DRBG instantiation using a different internal state than the DRBG used by the RBG3 construction may operate as either an RBG2(P) or RBG2(NP) construction.

B.2. Example of an RBG1 Construction

An RBG1 construction only has access to a randomness source during instantiation (i.e., when it is seeded; see Sec. 4). In Fig. 46, the DRBG used by the RBG1 construction and the randomness source reside in two different cryptographic modules with a physically secure channel connecting them during the instantiation process.

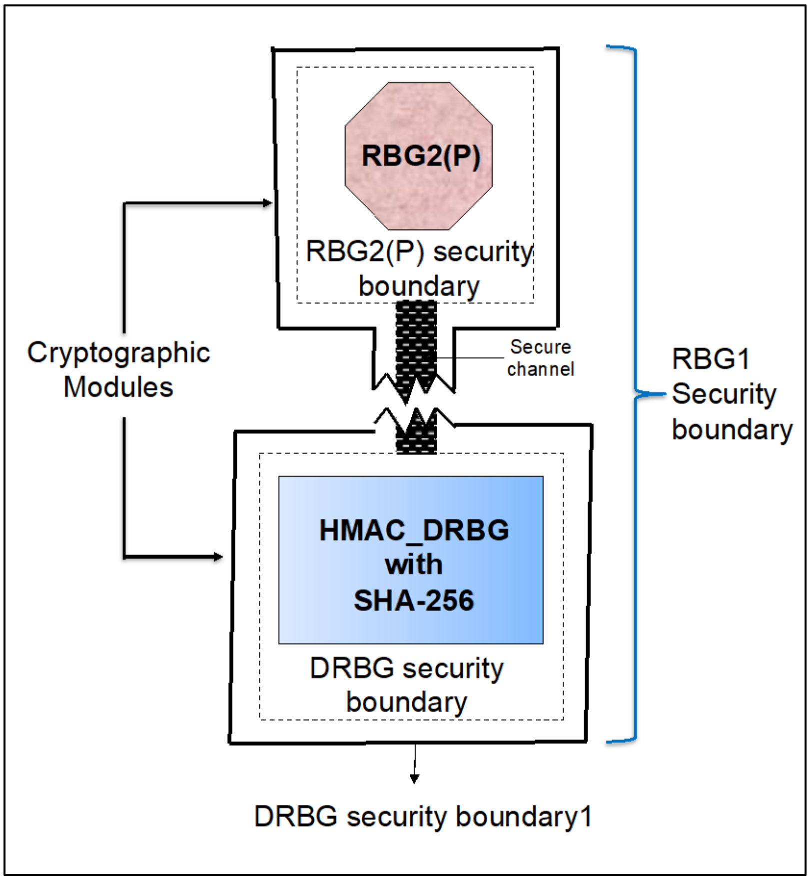

Fig. 46. Example of an RBG1 construction

Following DRBG instantiation, the secure channel is no longer available. For this example, the randomness source is an RBG2(P) construction (see Sec. 5) with a state handle of *RBG2_DRBG_state_handle*. The targeted security strength for the RBG1 construction is 256 bits, so a DRBG from SP 800-90A that is able to support this security strength must be used. HMAC_DRBG using SHA-256 is used in the example. A *personalization_string* is provided during instantiation, as recommended in Sec. 2.4.1.

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

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

B.2.1. Instantiation of the RBG1 Construction

A physically secure channel is required to transport the entropy bits from the randomness source (i.e., the RBG2(P) construction) to the HMAC_DRBG during instantiation; an example of an RBG2(P) construction is provided in Appendix B.4. After the instantiation of the RBG1 construction, the randomness source and the secure channel are no longer available.

1. The HMAC_DRBG is instantiated by an application when sending an instantiate request to the DRBG:

   (*status*, *RBG1_DRBG_state_handle*) = **DRBG_Instantiate_request**(256, "Device 7056"),

   where:

   - A security strength of 256 bits is requested for the HMAC_DRBG used in the RBG1 construction.
   - The *personalization string* to be used for this example is "Device 7056".

2. The **DRBG_Instantiate_request** results in the execution of the **DRBG_Instantiate** function within the DRBG of the RBG1 construction (see Sec. 2.8.1.1):

   (*status*, *RBG1_DRBG_state_handle*) = **DRBG_Instantiate**(256, "Device 7056").

3. The instantiate function sends a reseed request to the RBG2(P) construction (i.e., the randomness source; see requirement 18 in Sec. 4.4.1).

   status = **DRBG_Reseed_request**(*RBG2_DRBG_state_handle*),

   where *RBG2_DRBG_state_handle* is the state handle for the internal state in the RBG2(P) construction.

4. Upon receiving a reseed request, the RBG2(P) implementation executes a reseed function:

   status = **DRBG_Reseed**(*RBG2_DRBG_state_handle*).

   If an error is indicated by the returned *status*, the error is returned to the RBG1 construction by the RBG2(P) construction in response to the reseed request and forwarded to the application by the RBG1 construction in response to the instantiate request. The DRBG within the RBG1 construction has NOT been instantiated.

   Otherwise, a *status* of success is returned to the RBG1 construction in response to the reseed request (i.e., the DRBG within the RBG2(P) construction has been successfully reseeded).

5. Upon receiving a *status* of success in response to the reseed request, the RBG1 construction then sends a generate request to the RBG2(P) construction (see Sec. 5.2.2).

   (*status*, *seed_material*) = **DRBG_Generate_request**(*RBG2_DRBG_state_handle*, 384, 256),

   where 384 is the $3s/2$ bits needed to instantiate the HMAC_DRBG at a security strength of 256 bits.

6. Upon receiving a generate request, the RBG2(P) construction executes a generate function using information from the request:

   (*status, seed_material*) = **DRBG_Generate**(*RBG2_DRBG_state_handle*, 384, 256).

If an error is indicated by the returned *status*, the error is returned to the RBG1 construction by the RBG2(P) construction in response to the generate request and forwarded to the application by the RBG1 construction in response to the instantiate request. The DRBG within the RBG1 construction is NOT instantiated.

If a *status* of success is returned from the generate function, 384 bits of *seed_material* are also provided and sent to the RBG1 construction in response to the generate request.

7. The DRBG within the RBG1 construction uses the *seed_material* provided by the RBG2(P) construction and the *personalization_string* provided by the application in the instantiate request (see step 1) to create the seed to instantiate the DRBG (see SP 800-90A).

   If the instantiation is not successful, an error is returned to the application in response to the instantiate request. The DRBG within the RBG1 construction has NOT been instantiated.

   If the instantiation is successful, the internal state is established. A *status* of SUCCESS and the *RBG1_DRBG_state_handle* are returned to the application requesting instantiation, 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 can be obtained as follows:

1. A consuming application sends a generate request to the RBG1 construction:

   (*status, returned_bits*) = **DRBG_Generate_request**(*RBG1_DRBG_state_handle, requested_number_of_bits, requested_security_strength*).

   - *RBG1_DRBG_state_handle* is returned as the state handle during instantiation (see Appendix B.2.1).
   - The *requested_security_strength* may be any value that is less than or equal to 256 (i.e., the instantiated security strength recorded in the DRBG's internal state).

2. Upon receiving a generate request, the RBG1 construction executes a generate function, as specified in Sec. 2.8.1.2:

   (*status, returned_bits*) = **DRBG_Generate**(*RBG1_DRBG_state_handle, requested_number_of_bits, requested_security_strength*).

   If an error is returned as the *status*, the RBG1 construction forwards the error indication to the application (in response to the generate request). *returned_bits* is a *Null* string.

   If an indication of success is returned as the *status*, the *requested_number_of_bits* are provided as the *returned_bits* to the consuming application in response to the generate request.

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 Fig. 47).

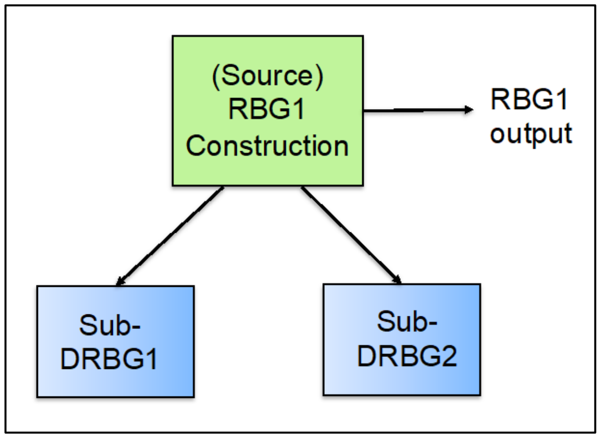

Fig. 47. Sub-DRBGs based on an RBG1 construction

The instantiation of the RBG1 construction is discussed in Appendix B.2. The RBG1 construction that is used as the randomness source includes an HMAC_DRBG and has been instantiated to provide a security strength of 256 bits. The state handle for the construction is *RBG1_DRBG_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 (i.e., HMAC_DRBG using SHA-256). Neither the RBG1 construction nor the sub-DRBGs can be reseeded.

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 (i.e., sub-DRBG1) that provides output for an application that requires a security strength of no more than 128 bits, and
- Access to a second sub-DRBG (i.e., 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 B.2.

B.3.1.1. Instantiating Sub-DRBG1

1. Sub-DRBG1 is instantiated when an application sends an instantiate request to the RBG1 construction:

   (*status*, *sub-DRBG1_state_handle*) =
   Instantiate_sub-DRBG_request(128, "Sub-DRBG App 1"),

   where

   - A security strength of 128 bits is requested for sub-DRBG1,
   - The *personalization string* to be used for sub-DRBG1 is "Sub-DRBG App 1", The comma is Nand
   - The returned state handle for sub-DRBG1 will be *sub-DRBG1_state_handle.*

2. Upon receiving the instantiate request, the RBG1 construction executes its instantiate function for a sub-DRBG (see Sec. 4.3.1):

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

   As specified for the **Instantiate_sub-DRBG** function, the DRBG in the RBG1 construction will attempt to generate $3s/2 = 192$ bits of seed material and combine it with "Sub-DRBG App 1" (i.e., the personalization string) to create a seed for the internal state of sub-DRBG1.

   If an error is returned as the *status*, the RBG1 construction forwards the error indication to the application in response to the instantiate request. The sub-DRBG is NOT instantiated.

   If an indication of success is returned as the *status*, the RBG1 construction forwards the *status* to the application in response to the instantiate request. Sub-DRBG1 can now be requested directly to generate output. See Appendix B.3.2.

B.3.1.2. Instantiating Sub-DRBG2

Sub-DRBG2 is instantiated in the same manner as sub-DRBG1 but at a security strength of 256 bits and with a different personalization string.

1. The application sends an instantiate request to the RBG1 construction:

   (*status*, *sub-DRBG2_state_handle*) =
   Instantiate_sub-DRBG_request(256, "Sub-DRBG App 2").

2. The RBG1 construction executes an instantiate function for a sub-DRBG:

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

The DRBG in the RBG1 construction will attempt to generate $3s/2 = 384$ bits of seed material and combine it with "Sub-DRBG App 2" to create a seed for the internal state of sub-DRBG2.

If an error is returned as the *status*, the RBG1 construction forwards the error indication to the application in response to the instantiate request. The sub-DRBG is NOT instantiated.

If an indication of success is returned as the *status*, the RBG1 construction forwards the *status* to the application in response to the instantiate request. Sub-DRBG2 can now be requested directly to generate output. See Appendix B.3.2.

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 can be requested from the sub-DRBG by a consuming application.

1. An application sends the following generate request:

   (*status, returned_bits*) = **DRBG_Generate_request**(*sub-DRBG_state_handle, requested_number_of_bits, requested_security_strength*),

   - For sub_DRBG1, *sub-DRBG_state_handle* = *sub-DRBG1_state_handle*.
   - For sub-DRBG2, *sub-DRBG_state_handle* = *sub-DRBG2_state_handle*.
   - *requested_number_of_bits* must be ≤ 2^{19} (see SP 800-90A for the HMAC_DRBG parameters).
   - For sub_DRBG1, security strength must be ≤ 128.
   - For sub_DRBG2, security strength must be ≤ 256.

2. The sub-DRBG executes the generate request (see Sec. 2.8.1.2):

   (*status, returned_bits*) = **DRBG_Generate**(*sub-DRBG_state_handle, requested_number_of_bits, security_strength*).

   If an error is returned as the *status*, the sub-DRBG forwards the error indication to the application in response to the generate request. The *returned_bits* string is *Null*.

   If an indication of success is returned as the *status*, the sub-DRBG forwards the *status* to the application along with the requested number of newly generated bits.

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

For this example of an RBG2(P) construction, no conditioning function is used, and only a single DRBG instantiation will be used (see Fig. 48), so a state handle is not needed. A physical and a non-physical entropy source are used. Full-entropy output is not provided by the entropy sources.

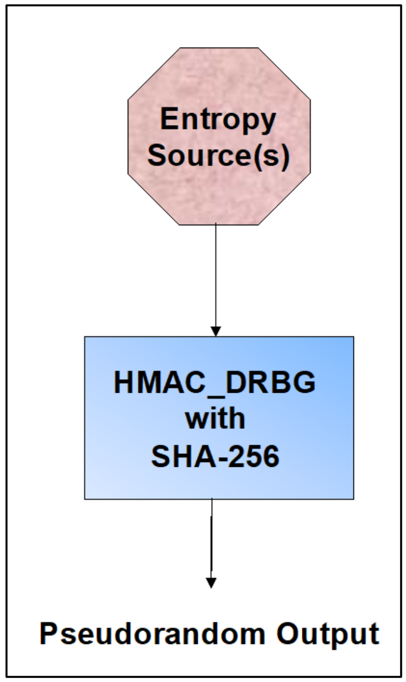

Fig. 48. Example of an RBG2 construction

The targeted security strength is 256 bits, so a DRBG from SP 800-90A 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 Sec. 2.4.1. Reseeding is supported and will be available on demand. Method 1 is used for counting the entropy produced by the entropy sources (i.e., only entropy from the physical entropy source is counted).

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(P) Construction

1. The RBG2(P) construction is instantiated by an application using an instantiate request:

   status = **DRBG_Instantiate_request**(256, “RBG2 42”).

   Since there is only a single instantiation, a *state_handle* is not used for this example. The *personalization string* to be used for this example is “RBG2 42”.

2. Upon receiving the instantiate request, the RBG2(P) construction executes an instantiate function:

   status = **DRBG_Instantiate**(256, “RBG2 42”).

The seed material for establishing the security strength (s) of the DRBG (i.e., $s = 256$ bits) is requested using the following call to the entropy source (see Sec. 2.8.2 and item 2 in Sec. 5.2.1):

(*status*, *seed_material*) = **Get_entropy_bitstring**(384, *Method_1*),

where $3s/2 = 384$ bits of entropy are requested from the entropy source, and Method 1 is used to count only the entropy produced by the physical entropy source.

If *status* = SUCCESS is returned in response to the **Get_entropy_bitstring** call, the HMAC_DRBG is seeded using *seed_material* and the *personalization_string* ("RBG2 42"). The internal state is recorded (including the security strength of the instantiation), and *status* = SUCCESS is returned to the consuming application in response to the instantiation request.

If the *status* returned in response to the **Get_entropy_bitstring** call indicates an error, then the internal state is not created, the *status* is returned to the consuming application in response to the instantiation request, and the RBG cannot be used to generate bits.

B.4.2. Generation Using an RBG2(P) Construction

Assuming that the RBG has been successfully instantiated (see Appendix B.4.1):

1. Pseudorandom bits can be requested from the RBG by a consuming application:

   (*status, returned_bits*) = **DRBG_Generate_request**(*requested_number_of_bits, requested_security_strength*).

   - Since there is only a single instantiation of the HMAC_DRBG, a *state_handle* was not returned from the **DRBG_Instantiate** (see Appendix B.4.1) and is not used during the generate request.
   - The *requested_security_strength* may be any value that is ≤ 256 (i.e., the instantiated security strength recorded in the HMAC_DRBG's internal state).

2. Upon receiving the generate request, the RBG executes the generate function (see Sec. 2.8.1.2):

   (*status, returned_bits*) = **DRBG_Generate**(*requested_number_of_bits, security_strength*).

   A *status* indication is returned to the requesting application in response to the **DRBG_Generate** call. If *status* = SUCCESS, a bitstring of at least *requested_number_of_bits* is provided as the *returned_bits*. If *status* = FAILURE, *returned_bits* is an empty bitstring.

B.4.3. Reseeding an RBG2(P) Construction

The HMAC_DRBG will be reseeded 1) if explicitly requested by the consuming application or 2) automatically during a **DRBG_Generate** call at the end of the DRBG's designed *seedlife* (see the **DRBG_Generate** function specification in SP 800-90A and Sec. 5.2.3 herein).

1. An application may request a reseed of the DRBG using a reseed request:

   status = DRBG_Reseed_request().

   Since there is only a single instantiation of the HMAC_DRBG, a *state_handle* was not returned from the **DRBG_Instantiate** function (see Appendix B.4.1) and is not used during the reseed request.

2. Upon receiving the reseed request or when the end of the seedlife is determined, the RBG executes the reseed function (see Sec. 2.8.1.3):

   status = DRBG_Reseed().

   The **DRBG_Reseed** function uses a **Get_randomness-source_input** call to access the entropy source.

   (*status*, *seed_material*) = **Get_entropy_bitstring**(256, *Method_1*).

   Method_1 indicates that only the entropy from the physical entropy source should be counted.

   If *status* = SUCCESS is returned by **Get_entropy_bitstring**, the *seed_material* contains at least 256 bits of entropy and is at least 256 bits long. *Status* = SUCCESS is returned to the RBG2 construction in response to the **DRBG_Reseed** call, and the *status* is forwarded to the application in response to the reseed request, if appropriate.

   If the *status* indicates an error, *seed_material* is an empty (e.g., null) bitstring. The HMAC_DRBG is not reseeded, the *status* is returned to the **DRBG_Reseed** function in the RBG2 construction, and the *status* is then forwarded to the application in response to the reseed request, if appropriate. Depending on the error, the DRBG operation may be terminated (see item 10 in Sec. 2.6).

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

This construction is specified in Sec. 6.4 and requires a DRBG and a source of full-entropy bits. For this example, a single physical entropy source that does not provide full-entropy output is used, so the vetted hash conditioning function listed in SP 800-90B using SHA-256 is used as an external conditioning function. Since the type of entropy source is known, the counting method is known and need not be indicated when requesting entropy.

The Hash_DRBG specified in SP 800-90A 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 Fig. 49) 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.

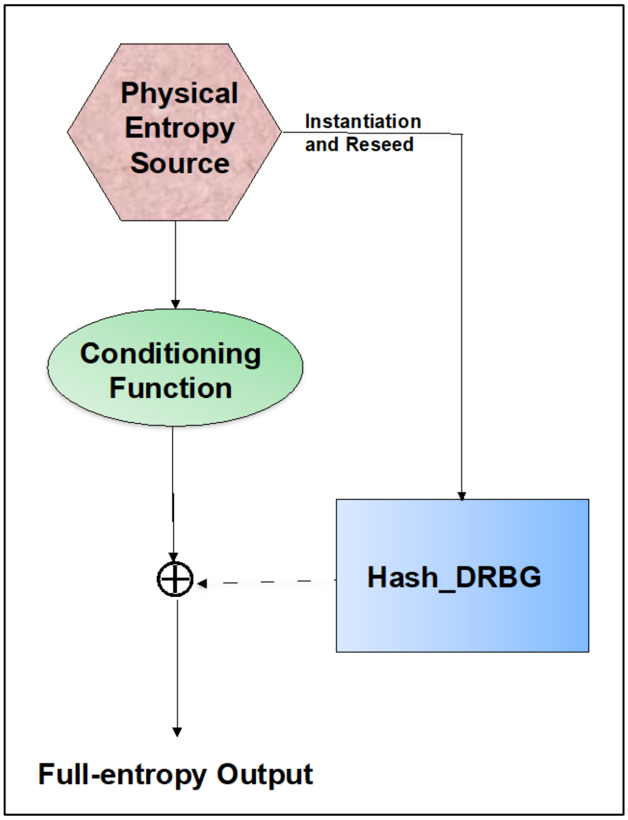

Fig. 49. Example of an RBG3(XOR) construction

The DRBG is instantiated and reseeded at a 256-bit security strength. In this example, only a single instantiation is used, and a personalization string is provided during instantiation. Calls are made to the RBG using the RBG3(XOR) calls specified in Sec. 6.4. 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

1. An application instantiates an RBG3(XOR) construction using an instantiate request that will instantiate the DRBG within the RBG:

status = **Instantiate_RBG3_DRBG_request**(256, "RBG3(XOR)").

Since only a single instantiation is used, there is no need for a state handle. The HMAC_DRBG is requested to be instantiated at a security strength of 256 bits using "RBG3(XOR)" as a personalization string.

2. Upon receiving an instantiate request, the RBG3(XOR) construction executes an instantiate function:

status = **RBG3(XOR)_Instantiate**(256, "RBG3(XOR)").

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

(*status*, *seed_material*) = **Get_entropy_bitstring**(384).

If *status* = SUCCESS is returned from the **Get_entropy_bitstring** call, the Hash_DRBG is seeded using the *seed_material* and the *personalization_string* (i.e., "RBG3(XOR)"). The internal state is recorded (including the 256-bit security strength of the instantiation), and *status* = SUCCESS is returned to the RBG3(XOR) construction and forwarded to the consuming application in response to the instantiate request (from step 1). The RBG can be used to generate full-entropy bits.

If the *status* returned from the **Get_entropy_bitstring** call indicates an error, the *status* is forwarded by the RBG3(XOR) construction to the consuming application. 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.5.1), the RBG can be called by a consuming application to generate output with full entropy.

B.5.2.1. Generation

1. An application requests the generation of full-entropy bits using:

(*status*, *returned_bits*) = **RBG3_DRBG_Generate_request**(*n*),

where *n* indicates the requested number of bits to generate. A state handle is not included since a state handle was not returned during instantiation (see Appendix B.5.1).

2. Upon receiving a generate request, the RBG3(XOR) construction executes a call to the generate function:

(*status*, *returned_bits*) = **RBG3(XOR)_Generate**(*n*).

The construction of the **RBG3(XOR)_Generate** function in Sec. 6.4.1.2 is used as follows:

RBG3(XOR)_Generate:

Input:

n: The number of bits to be generated.

Output:

status: The status returned by the **RBG3(XOR)_Generate** function.

returned_bits: The newly generated bits or a *Null* bitstring.

Process:

2.1 (*status, ES_bits*) = Get_conditioned_full-entropy_input(*n*).

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

2.3 (*status*, *DRBG_bits*) = **DRBG_Generate**(*n,* 256).

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

2.5 *returned_bits* = *ES_bits* ⊕ *DRBG_bits.*

2.6 Return (SUCCESS, *returned_bits*).

The *state_handle* parameter is not used in the **RBG3(XOR)_Generate** call or the **DRBG_Generate** function call (in step 2.3) for this example since a *state_handle* was not returned from the **RBG3(XOR)_ Instantiate** function (see Appendix B.5.1).

In step 2.1, 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, which are returned as the *ES_bits*.

Step 2.2 checks that the **Get_conditioned_full-entropy_input** call in step 2.1 was successful. If it was not successful, the **RBG3(XOR)_Generate** function is aborted, returning *status* ≠ SUCCESS and a *Null* bitstring to the RBG3(XOR) construction. The *status* and *Null* bitstring are then forwarded to the application in response to the generate request (in step 1).

Step 2.3 calls the Hash_DRBG to generate *n* bits at a security strength of 256 bits. The generated bitstring is returned as *DRBG_bits*.

Step 2.4 checks that the **DRBG_Generate** function invoked in step 2.3 was successful. If it was not successful, the **RBG3(XOR)_Generate** function is aborted, returning *status* ≠ SUCCESS and a *Null* bitstring to the RBG3(XOR) construction. The *status* and *Null* bitstring are then forwarded to the application in response to the generate request (in step 1).

If step 2.3 returns an indication of success, the *ES_bits* returned in step 2.1 and the *DRBG_bits* obtained in step 2.3 are XORed together in step 2.5. The result is returned to the RBG3(XOR) construction in step 2.6 and forwarded to the application in response to the generate request (in step 1).

B.5.2.2. Get_conditioned_full-entropy_input Function

The **Get_conditioned_full-entropy_input** procedure is specified in Sec. 3.2.2.2. For this example, the routine becomes the following:

Get_conditioned_full_entropy_input:

Input:

n: The number of full-entropy bits to be provided.

Output:

1. *status*: The status returned from the **Get_conditioned_full_entropy_input** function.
2. *Full-Entropy_bitstring*: The newly acquired *n*-bit string with full entropy or a *Null* bitstring.

Process:

1. *temp* = the *Null* string.
2. *ctr* = 0.
3. While *ctr* < *n*, do
   3.1 (*status, Entropy_bitstring*) = **Get_entropy_bitstring** (320).
   3.2 If (*status* ≠ SUCCESS), then return (*status*, *Null*).
   3.3 *conditioned_output* = $\textbf{Hash}_{\textbf{SHA\_256}}$(*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 (i.e., *output_len* + 64 bits, where *output_len* = 256 for SHA-256).
- Step 3.2 checks whether the *status* returned in step 3.1 indicated a success. If the *status* did not indicate a success, the *status* is returned to the **RBG3(XOR)_Generate** function (in Appendix B.5.2.1) along with a *Null* bitstring.
- Step 3.3 invokes the hash conditioning function (see Sec. 3.2.1.2) 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*).
- 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 (i.e., *Full-Entropy_bitstring*).

B.5.3. Reseeding an RBG3(XOR) Construction

The Hash_DRBG within the RBG3(XOR) construction 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 **DRBG_Generate** call (see SP 800-90A and step 2.3 in Appendix B.5.2.1).

The consuming application uses a reseed request to reseed the DRBG within the RBG3(XOR) construction:

status = **DRBG_Reseed_request**().

A state handle is not provided for this example since none was provided during instantiation.

Whether reseeding is done automatically during a **DRBG_Generate** call or is specifically requested by a consuming application, the **DRBG_Reseed** call for this example is:

status = **DRBG_Reseed**().

Again, a state handle is not provided since none was provided during instantiation.

A **Get_entropy_bitstring** call to the entropy source is used to obtain the entropy for reseeding:

(*status*, *seed_material*) **= Get_entropy_bitstring**(256).

If *status* = SUCCESS is returned by the **Get_entropy_bitstring** call, *seed_material* consists of at least 256 bits that contain at least 256 bits of entropy. These bits are used by the **DRBG_Reseed** function to reseed the Hash_DRBG. If the reseed was requested by an application, the *status* is returned to that application.

If the *status* indicates an error, the *seed_material* is a *Null* bitstring, and the Hash_DRBG is not reseeded. If the reseed was requested by an application, the error *status* is returned to the application.

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

This construction is specified in Sec. 6.5 and requires an entropy source and a DRBG, which is shown in the left half of Fig. 50 outlined in green with long dashes (– – – –). 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, which is shown in the right half of Fig. 50 outlined in blue with alternating dots and dashes (- • - • -).

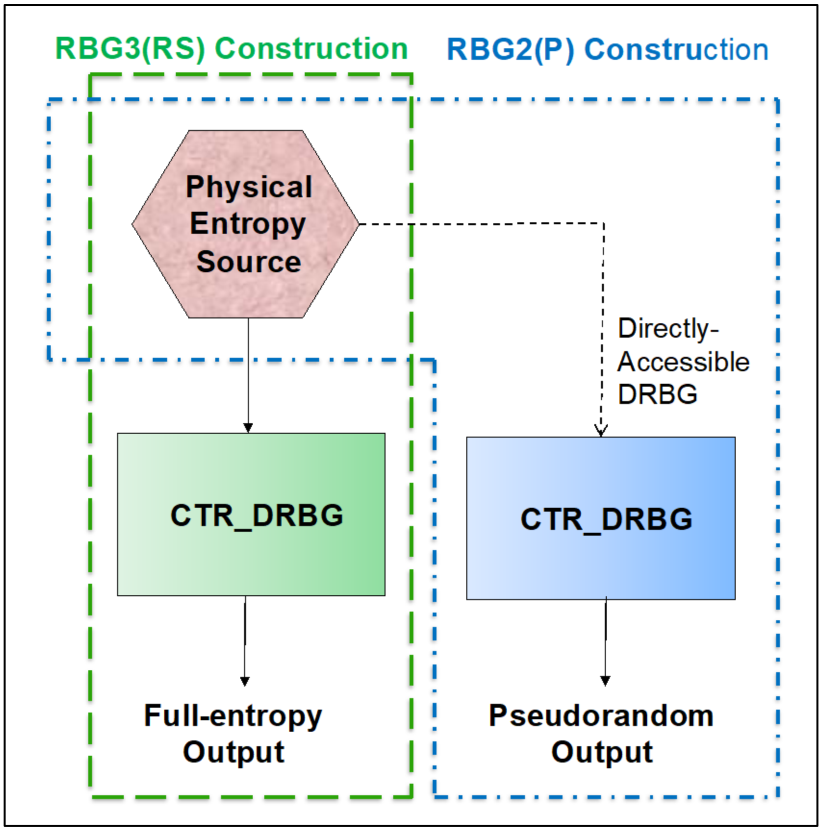

Fig. 50. Example of an RBG3(RS) construction

The CTR_DRBG specified in SP 800-90A 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 of the CTR_DRBG implementation. In this case, full-entropy output will not be required from the entropy source (see SP 800-90A).

As specified in Sec. 6.5, a DRBG used as part of the RBG must be instantiated and reseeded at a security strength of 256 bits when AES-256 is used in the DRBG.

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

Calls are made to the RBG3(RS) construction, as specified in Sec. 6.5. Calls made to the directly accessible DRBG (part of the RBG2(P) construction) use the RBG calls specified in Sec. 5. Since an entropy source is always available, the directly accessed DRBG can be reseeded.

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. Requirement 2 in Sec. 6.5.2 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, a global variable (*last_call*) within the RBG3(RS) security boundary is used to indicate whether the last use of the DRBG was as part of the RBG3(RS) construction or directly accessed:

- *last_call* = 1 if the DRBG was last used as part of the RBG3(RS) construction to provide full entropy output. If the next request is for generation by the DRBG directly, the DRBG must be reseeded before the requested output is generated.
- *last_call* = 0 otherwise. A reseed of the DRBG when accessed directly is not necessary. When the DRBG is first instantiated with entropy, *last_call* is set to zero.

See SP 800-90Ar1 for information about the internal state of the CTR_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 (i.e., 256 bits) if the entropy source has an undetected failure,
- Direct access to the DRBG 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
- On-demand reseeding of the DRBG (e.g., to provide prediction resistance for requests to 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.

1. An application requests the instantiation of the RBG3(RS) construction using:

   (*status*, *RBG3_DRBG_state_handle*) = **Instantiate_RBG3_DRBG_request**("RBG3(RS) 2024"),

   which requests the instantiation of the DRBG within the RBG3(RS) construction using "RBG3(RS) 2024" as the personalization string. In this example, the request does not include an indication of the security strength to be instantiated that would need to be checked against the security strength implemented for the DRBG (see Sec. 2.8.3.1 for a discussion).

2. Upon receiving the request, the RBG3(RS) construction executes the instantiate function:

   (*status*, *RBG3_DRBG_state_handle*) = **RBG3(RS)_ Instantiate**("RBG3(RS) 2024").

For this example, the **RBG3(RS)_Instantiate** function (see Sec. 6.5.1.1) in the DRBG includes an additional step to set the initial value of *last_call* to zero (i.e., if the first use of the DRBG is for direct access, a reseed of the DRBG before generating bits is not required). Setting the initial value of *last_call* is an implementation decision, but some method for this process is required:

2.1 (*status, RBG3_DRBG_state_handle*) = **DRBG_Instantiate**(*personalization_string*).

2.2 *last_call* = 0.

2.3 Return(*status*, *RBG3_DRBG_state_handle*).

In step 2.1, the **DRBG_Instantiate** function is used to instantiate the CTR_DRBG using "RBG3(RS) 2024" as the personalization string. Since the required security strength is known (i.e., 256 bits) and a derivation function is used in the CTR_DRBG implementation, the required entropy ($s + 128 = 384$ bits) is obtained from the entropy source using:

(*status*, *seed_material*) = **Get_entropy_bitstring**($s + 128$).

The *seed_material* and personalization string are used to seed the CTR_DRBG. Since the entropy source is known to be a physical entropy source, the counting method is known and not included as an input parameter.

Step 2.2 sets *last_call* = 0 so that if the initial request is for direct access to the DRBG, a reseed will not be initially required before generating bits (i.e., entropy has just been acquired as a result of the instantiation process).

In step 2.3, the *status* and the state handle for the DRBG's internal state are returned to the **RBG3(RS)_Instantiate** function and forwarded to the application in response to the instantiate request in step 1.

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

Assuming that the DRBG in 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.

1. An application requests the generation of full-entropy bits using:

   (*status, returned_bits*) = **RBG3_ Generate_request**(*RBG3_DRBG_state_handle, n*),

where *RBG3_DRBG_state_handle* was provided during DRBG instantiation (see Appendix B.6.1), and *n* is the number of requested bits.

2. Upon receiving the generate request, the RBG3(RS) construction executes the generate function (see Sec. 6.5.1.2.1):

   (*status, returned_bits*) = **RBG3(RS)_Generate**(*RBG3_DRBG_state_handle, n*).

   A few modifications to the **RBG3(RS)_Generate** function have been made, resulting in the following:

   RBG3(RS)_ Generate:

   Input:

- *RBG3_DRBG_state_handle*: The state handle for the DRBG's internal state (see Appendix B.6.1).
- *n*: The number of full-entropy bits to be generated.

Output:

- *status*: The status returned from the **RBG3(RS)_Generate** function.
- *returned_bits*: The newly generated bits or a *Null* bitstring.

Process:

2.1 *temp* = *Null*.

2.2 *sum* = *0*.

2.3 While ($sum < n$),

  2.3.1 *status* = **DRBG_Reseed**(*RBG3_DRBG_state_handle*).

  2.3.2 If ($status \neq$ SUCCESS), then return (*status, Null*).

  2.3.3 (*status*, *full_entropy_bits* = **DRBG_Generate**(*RBG3_DRBG_state_handle*, 256).

  2.3.4 If ($status \neq$ SUCCESS), then return (*status*, *Null*).

  2.3.5 *temp* = *temp* || *full_entropy_bits*.

  2.3.6 $sum = sum + s$.

2.4 *last_call* = 1.

2.5 Return (SUCCESS, **leftmost**(*temp*, *n*)).

Steps 2.1 and 2.2 initialize *temp* to a *Null* string for accumulating the requested output and *sum* to zero for counting the entropy generated.

Step 2.3 generates the requested output with full entropy.

Step 2.3.1 reseeds the DRBG. Whenever the RBG3(RS) construction is requested to generate bits, the DRBG is always reseeded with $s + 64 = 320$ bits directly from the entropy source (see Appendix B.6.4).

Step 2.3.2 checks the *status* of the reseed process and returns the *status* and a *Null* string if the reseed process was not successful.

Step 2.3.3 requests the generation of 256 bits.

Step 2.3.4 checks the *status* of the generate process and returns the *status* and a *Null* string if the generate process was not successful. The "256" could be omitted since it is known to be the same as the hard-coded security strength.

Step 2.3.5 assembles the full-entropy bitstring.

Step 2.3.6 counts the number of bits assembled so far.

In step 2.4, the *last_call* value is set to one to indicate that the requested bits were generated by the RBG3(RS) construction rather than by direct use of the DRBG.

3. The *status* and generated bits from the **RBG3(RS)_Generate** function in step 2 are returned to the RBG3(RS) construction and forwarded to the application in response to the generate request in step 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_DRBG_state_handle* obtained during instantiation (see Appendix B.6.1). Pseudorandom bits can be generated directly by the CTR_DRBG as follows:

1. An application requests the generation of pseudorandom bits directly from the DRBG within the RBG3(RS) construction:

   (*status, returned_bits*) = **DRBG_Generate_request**(*RBG3_DRBG_state_handle, n, s*),

   where *RBG3_DRBG_state_handle* was obtained during instantiation (see Appendix B.6.1), *n* is the requested number of bits to be returned, and *s* is the requested security strength.

2. Upon receiving the generate request, the RBG3(RS) construction executes a **DRBG_Generate** function rather than an **RBG3(RS)_Generate** function:

   (*status, returned_bits*) = **DRBG_Generate**(*RBG3_DRBG_state_handle, n*).

   The security strength parameter (i.e., 256) is omitted since its value has been hard-coded.

   The **DRBG_Generate** function specified in SP 800-90A has been modified to determine whether a reseed is required before generating the requested output by checking the value of *last_call*. An extraction[31] of the **DRBG_Generate** function in SP 800-90A is:

   [After other preliminary checks have been performed]

   If ((*last_call* = 1) OR (*reseed_counter* > *reseed_interval*)), then

   status = **DRBG_Reseed**(*RBG3_DRBG_state_handle*).

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

   ...

   (*returned_bits, new_working_state_values*) =
   Generate_algorithm(*current_working_state_values, requested_number_of_bits*).

   last_call = 0.

   [Closing steps to update the internal state]

[31] The complete **DRBG_Generate** function is significantly longer.

An additional step has also been included above to indicate that this use of the DRBG is direct rather than part of the RBG3(RS) construction (i.e., setting $last\_call = 0$). This step is used to indicate that if the next use of the DRBG is also by direct access, a reseed is not required before generating bits.

B.6.4. Reseeding a DRBG

When operating as part of the RBG3(RS) construction, the **DRBG_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 (see Sec. 6.5.1.3).

When operating as the directly accessible DRBG, the DRBG is reseeded 1) if explicitly requested by the consuming application, 2) whenever the previous use of the DRBG was by the **RBG3(RS)_Generate** function (see Appendix B.6.2), or 3) automatically during a **DRBG_Generate** call at the end of the seedlife of the RBG2(P) construction (see the **DRBG_Generate** function specification in SP 800-90A).

1. The reseed function is requested by an application using:

   status = **DRBG_Reseed_request**(*RBG3_DRBG_state_handle*),

   where *RBG3_DRBG_state_handle* was obtained during instantiation.

2. The **DRBG_Reseed** function is executed in response to a reseed request by an application (see step 1) or during the generation process (see Appendices B.6.2 and B.6.3):

   status = **DRBG_Reseed**(*RBG3_DRBG_state_handle*).

   For this example, the **DRBG_Reseed** function is modified to obtain $s + 64$ bits of entropy rather than the "normal" s bits of entropy (see method A for step 3.1 in Sec. 6.5.1.2.1).

   (*status, seed_material*) = **Get_entropy_bitstring**($s + 64$).

   If *status* = SUCCESS is returned by the **DRBG_Reseed** function, the internal state has been updated with at least 320 bits of fresh entropy (i.e., $256 + 64 = 320$). *Status* = SUCCESS is returned to the calling application by the **DRBG_Reseed** function.

   If *status* ≠ SUCCESS (e.g., the entropy source has failed), the DRBG has not reseeded, and an error indication is returned as the status from **DRBG_Reseed** function to the calling application.

B.7. DRBG Chains Using the RBGC Construction

A chain of DRBGs consists of RBGC constructions and an initial randomness source on the same computing platform. For this example, the initial randomness source is a physical entropy source that provides output with full entropy (i.e., the initial randomness source is a full-entropy source). The chain includes two RBGC constructions: the root RBGC construction ($RBGC_1$) and a child ($RBGC_2$) (see Fig. 51).

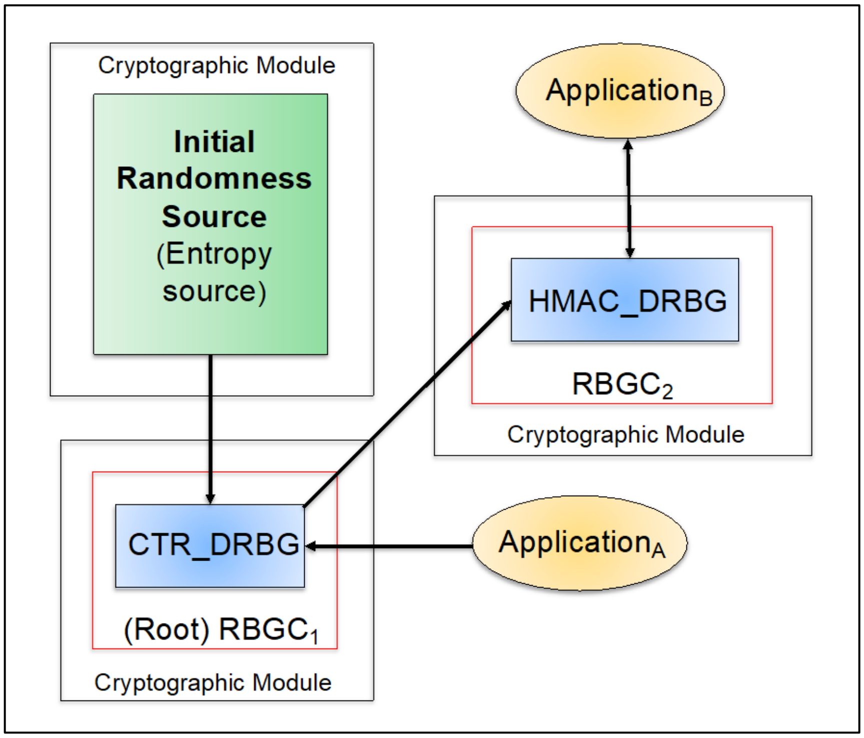

Fig. 51. Example of DRBG chains

In this example, a CTR_DRBG with no derivation function is used in the root ($RBGC_1$). It will be seeded and reseeded at a security strength of 192 bits using the initial randomness source.

$RBGC_2$ is implemented using SHA-256 and the HMAC_DRBG. $RBGC_2$ will be seeded and reseeded at a security strength of 128 bits using the root ($RBGC_1$) as its randomness source.

B.7.1. Instantiation of the RBGC Constructions

The DRBG in each RBGC construction is instantiated by an application using a known randomness source, starting with the instantiation of the DRBG in the root using the initial randomness source (see Appendix B.7.1.1). Subsequent layers in the chain can be instantiated when an already-instantiated RBGC construction is available. For this example, after the root has been instantiated, the DRBG in a child RBGC construction ($RBGC_2$) can be instantiated using the root as its randomness source (see Sec. 7.2.1.2).

B.7.1.1. Instantiation of the Root RBGC Construction

The root of the DRBG chain is instantiated using the initial randomness source, which for this example is an entropy source that provides output with full entropy. The instantiation is requested by an application (i.e., $Application_A$ in Fig. 51). The CTR_DRBG in the root is implemented using AES-192, so a maximum security strength of 192 bits can be instantiated.

1. The application ($Application_A$) sends an instantiate request to the root requesting that the DRBG within the root be instantiated at a security strength of 192 bits:

   (*status*, *Root_DRBG_state_handle*) =
   DRBG_Instantiate_request(192, "Root RBGC"),

   where "Root RBGC" is the personalization string, and *Root_DRBG_state_handle* is the name of the state handle to be assigned to the internal state of the root's DRBG.

2. Upon receiving the instantiate request, the root ($RBGC_1$) executes the instantiate function for its DRBG:

   (*status, Root_DRBG_state_handle*) = **DRBG_Instantiate**(192, "Root RBGC").

   The **DRBG_Instantiate** function in the root determines that its DRBG (CTR_DRBG) needs to obtain 192 + 128 = 320 bits with full entropy from the full-entropy source. The root sends a **Get_entropy_bitstring** request to the randomness source to obtain 320 bits of seed material:

   (*status*, *seed_material*) = **Get_entropy_bitstring**(320, *Method_1*).

   Method_1 indicates that only entropy from a physical entropy source is to be counted.

   If the *status* indicates success and *seed_material* is returned from the initial randomness source (i.e., the full-entropy source), the CTR_DRBG is seeded using the *seed_material* and the *personalization_string* (i.e., "Root RBGC") (see SP 800-90A). The internal state is recorded (including the security strength of the instantiation), and the *status* and a state handle are returned to the root (RBC_1) and forwarded to the application in response to the instantiate request.

   If the *status* indicates an error, the internal state is not created. The *status* and an invalid state handle are returned to the root (RBC_1) and forwarded to the application in response to the instantiate request.

B.7.1.2. Instantiation of a Child RBGC Construction ($RBGC_2$)

A child RBGC construction can be instantiated by an application (i.e., $Application_B$ in Fig. 51) after the root has been successfully instantiated. In this example, the HMAC_DRBG in $RBGC_2$ is implemented using SHA-256, so a maximum security strength of 256 bits is possible. However, since the root RBGC construction (i.e., the randomness source for $RBGC_2$) can only support a security strength of 192 bits (see Appendix B.7.1.1), only requests for security strengths of 192 or 128 bits can be instantiated for $RBGC_2$.

The DRBG in $RBGC_2$ is instantiated as follows:

1. An application ($Application_B$) requests the instantiation of the DRBG in $RBGC_2$ at a security strength of 128 bits:

   (*status*, *RBGC2_DRBG_state_handle*) =
   DRBG_Instantiate_request(128, "RBGC2 DRBG"),

where "RBGC2 DRBG" is the personalization string, and *RBGC2_DRBG_state_handle* is the name of the state handle to be assigned to the DRBG in the $RBGC_2$ construction.

2. Upon receiving the instantiate request, the $RBGC_2$ construction executes the instantiate function for its DRBG:

   (*status, RBGC2_DRBG_state_handle*) = **DRBG_Instantiate**(128, "RBGC2 DRBG").

   The **DRBG_Instantiate** function in the DRBG sends a generate request to the root:

   (*status, seed_material*) = **DRBG_Generate**(*Root_DRBG_state_handle, 192, 128*),

   where

   - *Root_DRBG_state_handle* is the state handle for the internal state of the DRBG in the root (see Sec. 7.1.1).
   - The requested security strength is 128 bits, so for the HMAC_DRBG in $RBGC_2$, the number of bits requested from the root (i.e., $RBGC_2$'s randomness source) is $3s/2 = 192$ bits.

   See Appendix B.7.2 for the handling of a generate request by an RBGC construction.

   If the *status* returned from the randomness source ($RBGC_1$) in response to the generate request indicates a success, the HMAC_DRBG in $RBGC_2$ is seeded using the *seed_material* returned from the generate request (Appendix B.7.2) and the *personalization_string* ("RBGC2 DRBG") from the instantiate request in step 1 (see SP 800-90A). The internal state is recorded (including the security strength of the instantiation), and the *status* and the state handle are returned to the $RBGC_2$ construction to be forwarded to the application that requested the instantiation of the DRBG in the $RBGC_2$ construction.

   If the *status* indicates an error, then the internal state is not created. The *status* and an invalid state handle are returned to the $RBGC_2$ construction to be forwarded to the application that requested the instantiation of the DRBG in the $RBGC_2$ construction.

B.7.2. Requesting the Generation of Pseudorandom Bits

1. An application or a child RBGC construction requests the generation of pseudorandom bits as follows:

   (*status, seed_material*) = **DRBG_Generate_request**(*DRBG_state_handle, n, s*),

   where

   - *DRBG_state_handle* is the state handle for the internal state of the DRBG in the RBGC construction requested to generate the bits. For this example, the state handle is *Root_DRBG_state_handle* for the DRBG in the root RBGC construction. For $RBGC_2$, the state handle is *RBGC2_DRBG_state_handle*.
   - *n* is the number of bits to be generated using the DRBG in the RBGC construction.

- *s* is the required security strength to be supported by the DRBG in the RBGC construction.

2. Upon receiving the generate request, the RBGC construction executes the generate function for its DRBG:

   (*status, seed_material*) = **DRBG_Generate**(*DRBG_state_handle, n, s*).

   If the returned *status* indicates success, the requested number of bits are returned (*seed_material*) to the RBGC construction and forwarded to the requesting entity with the *status*. The requesting entity is either an application or a child of the RBGC construction.

   If the returned status indicates an error, *seed_material* is a *Null* bitstring. This could, for example, be the result of requesting a higher security strength than is instantiated for the DRBG requested to generate bits. The *status* and the *Null* bitstring are returned to the RBGC construction and forwarded to the requesting entity.

B.7.3. Reseeding an RBGC Construction

The DRBG in an RBGC construction may be explicitly requested to be reseeded by an application, or the DRBG may automatically reseed itself (e.g., at the end of its seedlife or after some system interrupt).

1. An application requests the reseed of the DRBG in an RBGC construction as follows:

   (*status*) = **DRBG_Reseed_request**(*DRBG_state_handle*).

   DRBG_state_handle is *Root_DRBG_state_handle* for $RBGC_1$ and *RBG2_DRBG_state_handle* for $RBGC_2$.

2. Upon receiving a reseed request or if scheduled for automatic reseeding, the RBGC construction executes the reseed function for its DRBG:

   status = **DRBG_Reseed**(*DRBG_state_handle*).

Appendix B.7.3.1 discusses the reseed function in the root's DRBG, and Appendix B.7.3.2 discusses the reseed function in the DRBG of $RBGC_2$.

B.7.3.1. Reseeding the Root RBGC Construction

The **DRBG_Reseed** function in the root uses the initial randomness source to reseed in the same manner as for instantiation (i.e., by sending a **Get_entropy_bitstring** request to the entropy source). For the CTR_DRBG in the root, 320 bits are again requested:

(*status*, *seed_material*) **= Get_entropy_bitstring**(320, *Method_1*).

If the returned *status* indicates a success, *seed_material* is returned from the initial randomness source, and the CTR_DRBG within the root is reseeded using the *seed_material* (see SP 800-90A). The DRBG's internal state is updated, and the *status* is returned to the application by the **DRBG_Reseed** function in the root RBGC construction.

If the *status* indicates an error, then the internal state is not updated. The *status* is returned to the application.

B.7.3.2. Reseeding a Child RBGC Construction

The **DRBG_Reseed** function in the RBGC construction uses its randomness source in the same manner as for instantiation (i.e., by sending a **DRBG_Generate_request** to its randomness source, which is the root in this example).

For the HMAC_DRBG in $RBGC_2$, $s = 128$ bits are requested from the root RBGC construction (where s is the security strength of the DRBG instantiation in $RBGC_2$; see Appendix B.7.1.2).

(*status, seed_material*) = **DRBG_Generate**(*Root_DRBG_state_handle, 128, 128*),

where:

- *Root_DRBG_state_handle* is the state handle for the internal state of the DRBG in the root (see Appendix B.7.1.1).
- The requested security strength is 128 bits, so for the HMAC_DRBG in $RBGC_2$, the number of bits requested from the root ($RBGC_2$'s randomness source) is $s = 128$ bits.

Appendix B.7.2 discusses the handling of a generate request by an RBGC construction.

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

The derivation functions in this appendix will be included in the next revision of SP 800-90A along with other changes that are needed for consistency with this version of SP 800-90C.

C.1. Background and Scope

The CTR_DRBG, specified in SP 800-90A, uses the AES block cipher in FIPS 197 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 seed material 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 seed material (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 (i.e., 128 bits for AES). 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

$$seedlen = \text{the security strength} + \text{the block length.}$$

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

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 use 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 SP 800-38B. 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, and C_3 be 128-bit blocks defined as 000000...0, 101010...10, and 010101...01, respectively.
2. Get *df_Key*. Comment: See Appendix C.3.1.
3. Z = the Null string.
4. For $i = 1$ to B:

   $Z = Z \parallel \text{CMAC}(df\_Key, C_i \parallel input\_string)$.
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, and C_3 be 128-bit blocks defined as 000000...0, 101010...10, and 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 FIPS 197.

    5.4 $Z = Z \| V$.

6. Z = **leftmost**(Z, *number_of_bit_to_return*).
7. Return(Z).

Appendix D. List of Abbreviations and Acronyms

AES
Advanced Encryption Standard[32]

CAVP
Cryptographic Algorithm Validation Program

CMVP
Cryptographic Module Validation Program

DRBG
Deterministic Random Bit Generator[33]

FIPS
Federal Information Processing Standard

MAC
Message Authentication Code

NIST
National Institute of Standards and Technology

RBG
Random Bit Generator

SP
(NIST) Special Publication

Sub-DRBG
Subordinate DRBG

TDEA
Triple Data Encryption Algorithm[34]

XOR
Exclusive-Or (operation)

D.1. List of Symbols

$\mathbf{0}^x$
A string of x zeroes.

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

ε
A positive constant that is assumed to be smaller than 2^{-32}.

---

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

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.

X* ⊕ *Y
Boolean bitwise exclusive-or (also bitwise addition modulo 2) of two bitstrings X and Y of the same length.

+
Addition over real numbers.

X || Y
The concatenation of two bitstrings X and Y.

Appendix E. Glossary

additional input
Optional additional information that could be provided in a generate or reseed request by a consuming application.

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

alternative randomness source
A sibling of the parent randomness that may be used by a non-root RBGC construction for reseeding when the parent randomness source is unavailable.

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.

computing platform
A system's hardware, firmware, operating system, and all applications and libraries executed by that operating system. Components that communicate with the operating system through a peripheral bus or a network, either physical or virtual, are not considered to be part of the same computing platform.

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

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

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

DRBG chain
A chain of DRBGs in which one DRBG is used to provide seed material for another DRBG.

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

> *Note1:* The entropy of a random variable X is a mathematical measure of the amount of information gained by an observation of X.

> *Note2:* 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 8-bit output sample).

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

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

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

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

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}.

full-entropy source
An SP 800-90B-compliant entropy source that has been validated as providing output with full entropy or the validated combination of an SP 800-90B-compliant entropy source and an external conditioning function that provides full-entropy output.

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

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

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

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

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 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.

initial randomness source
The randomness source for the root RBGC construction in a DRBG chain of RBGC constructions.

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 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 to indicate a requirement that may not be testable by a CMVP testing lab.

Note: **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.

parent randomness source
The randomness source used to seed a non-root RBGC construction.

personalization string
An optional input value to a DRBG during instantiation.

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.

physically secure channel
A physical trusted and safe communication link established between an implementation of an RBG1 construction and its randomness source to securely communicate unprotected seed material without relying on cryptography. A physically secure channel protects against eavesdropping as well as physical or logical tampering by unwanted operators/entities, processes, or other devices between the endpoints.

prediction resistance
For a DRBG, 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 from an entropy source or RBG3 construction. See SP 800-90A for a more complete discussion. (Contrast with *backtracking resistance*.)

For an RBG, compromising the output of the RBG does not allow future outputs of the RBG to be predicted.

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
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 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.

> Note: An RBG2 construction may be either an RBG2(P) or RBG2(NP) construction.

RBG2(NP) construction
A non-physical 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 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*.

RBGC construction
An RBG construction used within a DRBG chain in which one DRBG is used to provide seed material for another DRBG. The construction does not provide full-entropy output.

reseed
To refresh the internal state of a DRBG with seed material from a randomness source.

root RBGC construction
The first RBGC construction in a DRBG chain of RBGC constructions.

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

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 the required DRBG functions and the DRBG's internal state.

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
Verb: To initialize or update the internal state of a DRBG with seed material and (optionally) a personalization string or additional input. The seed material should contain sufficient randomness to meet security requirements.

Noun: The combination of seed material and (optional) personalization or additional input.

seed material
An input bitstring from a randomness source that provides an assessed minimum amount of randomness (e.g., entropy) for a DRBG.

seedlife
The period of time between instantiating or reseeding a DRBG with seed material and either reseeding the DRBG with seed material containing new, unused randomness or uninstantiating the DRBG.

shall
The term used to indicate a requirement that is testable by a testing lab. See *testable requirement*.

Note: **Shall** may be coupled with **not** to become **shall not**.

should
The term used to indicate an important recommendation. Ignoring the recommendation could result in undesirable results.

Note: **Should** may be coupled with **not** to become **should not**.

sibling (randomness source)
A sibling of the parent randomness source for a non-root RBGC construction (i.e., the sibling can be considered the “aunt” or “uncle” in “human family” terms). The “grandparent” of the non-root RBGC construction is the parent of both the parent randomness source and the sibling.

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 and contained within the same security boundary as the 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, 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.