



Check for
updates

**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>

**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

Certain equipment, instruments, software, or materials, commercial or non-commercial, are identified in this paper in order to specify the experimental procedure adequately. Such identification does not imply recommendation or endorsement of any product or service by NIST, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

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

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

Authority

This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Modernization Act (FISMA) of 2014, 44 U.S.C. § 3551 et seq., Public Law (P.L.) 113-283. NIST is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130.

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

NIST Technical Series Policies

[Copyright, Use, and Licensing Statements](#)

[NIST Technical Series Publication Identifier Syntax](#)

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).

1 **Abstract**

2 The NIST Special Publication (SP) 800-90 series of documents supports the generation of high-
3 quality random bits for cryptographic and non-cryptographic use. SP 800-90A specifies several
4 deterministic random bit generator (DRBG) mechanisms based on cryptographic algorithms. SP
5 800-90B provides guidance for the development and validation of entropy sources. This
6 document (SP 800-90C) specifies constructions for the implementation of random bit generators
7 (RBGs) that include DRBG mechanisms as specified in SP 800-90A and that use entropy sources
8 as specified in SP 800-90B. Constructions for four classes of RBGs — namely, RBG1, RBG2, RBG3,
9 and RBGC — are specified in this document.

10 **Keywords**

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

14 **Reports on Computer Systems Technology**

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

25

26 **Note to Reviewers**

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

- 62 5. SHA-1 and the 224-bit hash functions (i.e., SHA-224, SHA-512/224, and SHA3-224) have
63 been removed from this version since NIST plans to disallow them after 2030 (see an
64 upcoming revision of SP 800-131A).
- 65 6. After the publication of SP 800-90C, SP 800-90A (Revision 1) will be revised to resolve
66 inconsistencies with this document. The revision will include:
- 67 • The **Instantiate_function**, **Generate_function**, and **Reseed_function** will be
68 renamed to **DRBG_Instantiate**, **DRBG_Generate**, and **DRBG_Reseed**. These
69 names have been used in SP 800-90C for clarity.
- 70 • The **Get_entropy_input** call discussed in SP 800-90Ar1 will be renamed to the more
71 general term "**Get_randomness-source_input**," which is used in SP 800-90C.
- 72 • SP 800-90Ar1 currently requires a nonce to be used during DRBG instantiation that is
73 either 1) a value with at least $(\text{security_strength}/2)$ bits of entropy or 2) a value that is
74 expected to repeat no more often than a $(\text{security_strength}/2)$ -bit random string
75 would be expected to repeat. The use of the nonce (as defined in SP 800-90Ar1) will
76 be replaced by additional bits provided by the randomness source.
- 77 • Parameters needed to use the DRBGs in the constructions specified in SP 800-90C will
78 be provided for each DRBG type in SP 800-90Ar1 (i.e., the Hash_DRBG,
79 HMAC_DRBG, and CTR_DRBG).
- 80 • Are there any other inconsistencies between this draft of SP 800-90C and the current
81 version of SP 800-90Ar1 at <https://doi.org/10.6028/NIST.SP.800-90Ar1>?

82

83 **Call for Patent Claims**

84 This public review includes a call for information on essential patent claims (claims whose use
85 would be required for compliance with the guidance or requirements in this Information
86 Technology Laboratory (ITL) draft publication). Such guidance and/or requirements may be
87 directly stated in this ITL Publication or by reference to another publication. This call also includes
88 disclosure, where known, of the existence of pending U.S. or foreign patent applications relating
89 to this ITL draft publication and of any relevant unexpired U.S. or foreign patents.

90 ITL may require from the patent holder, or a party authorized to make assurances on its behalf,
91 in written or electronic form, either:

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

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

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

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

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

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

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

109 **Table of Contents**

110	1. Introduction and Purpose	1
111	1.1. Audience	2
112	1.2. Document Organization	2
113	2. General Information.....	3
114	2.1. RBG Security.....	3
115	2.2. RBG Constructions	3
116	2.3. Sources of Randomness for an RBG.....	5
117	2.4. DRBGs.....	7
118	2.4.1. DRBG Instantiations.....	7
119	2.4.2. Reseeding, Prediction Resistance, and Compromise Recovery	8
120	2.5. RBG Security Boundaries.....	10
121	2.6. Assumptions and Assertions	11
122	2.7. General Implementation and Use Requirements and Recommendations.....	13
123	2.8. General Function Calls	14
124	2.8.1. DRBG Functions	15
125	2.8.1.1. DRBG Generation Request.....	17
126	2.8.1.2. DRBG Reseed	18
127	2.8.1.3. Get_randomness-source_input Call	19
128	2.8.2. Interfacing With Entropy Sources	19
129	2.8.3. Interfacing With an RBG3 Construction	20
130	2.8.3.1. Instantiating a DRBG Within an RBG3 Construction.....	20
131	2.8.3.2. Generation Using an RBG3 Construction.....	22
132	3. Accessing Entropy Source Output	24
133	3.1. Get_entropy_bitstring Process	24
134	3.2. External Conditioning.....	25
135	3.2.1. Conditioning Function Calls.....	26
136	3.2.1.1. Keys Used in External Conditioning Functions.....	27
137	3.2.1.2. Hash Function-based Conditioning Functions	27
138	3.2.1.3. Block Cipher-Based Conditioning Functions	28
139	3.2.2. Using a Vetted Conditioning Function.....	29
140	3.2.2.1. External Conditioning When Full Entropy is Not Required.....	29
141	3.2.2.2. Conditioning Function to Obtain Full-Entropy Bitstrings.....	31
142	4. RBG1 Construction Based on RBGs With Physical Entropy Sources.....	33

143	4.1. RBG1 Description	33
144	4.2. Conceptual Interfaces	34
145	4.2.1. Instantiating the DRBG in the RBG1 Construction	35
146	4.2.2. Requesting Pseudorandom Bits	38
147	4.3. Using an RBG1 Construction With Subordinate DRBGs (Sub-DRBGs)	39
148	4.3.1. Instantiating a Sub-DRBG	40
149	4.3.2. Requesting Random Bits From a Sub-DRBG	42
150	4.4. Requirements	42
151	4.4.1. RBG1 Construction Requirements	42
152	4.4.2. Sub-DRBG Requirements	43
153	5. RBG2 Constructions Based on Physical and/or Non-Physical Entropy Sources	45
154	5.1. RBG2 Description	45
155	5.2. Conceptual Interfaces	46
156	5.2.1. RBG2 Instantiation	46
157	5.2.2. Requesting Pseudorandom Bits From an RBG2 Construction	48
158	5.2.3. Reseeding an RBG2 Construction	49
159	5.3. RBG2 Construction Requirements	51
160	6. RBG3 Constructions Based on the Use of Physical Entropy Sources	53
161	6.1. General RBG3 Description	53
162	6.2. RBG3 Construction Types and Their Variants	54
163	6.3. General Requirements	54
164	6.4. RBG3(XOR) Construction	55
165	6.4.1. Conceptual Interfaces	56
166	6.4.1.1. Instantiation of the DRBG	56
167	6.4.1.2. Random Bit Generation by the RBG3(XOR) Construction	57
168	6.4.1.3. Pseudorandom Bit Generation Using a Directly Accessible DRBG	59
169	6.4.1.4. Reseeding the DRBG Instantiation	60
170	6.4.2. RBG3(XOR) Requirements	60
171	6.5. RBG3(RS) Construction	61
172	6.5.1. Conceptual Interfaces	61
173	6.5.1.1. Instantiation of the DRBG Within an RBG3(RS) Construction	61
174	6.5.1.2. Random and Pseudorandom Bit Generation	63
175	6.5.1.3. Reseeding	69
176	6.5.2. Requirements for an RBG3(RS) Construction	70

177	7. RBGC Construction for DRBG Chains	72
178	7.1. RBGC Description	72
179	7.1.1. RBGC Environment	72
180	7.1.2. Instantiating and Reseeding Strategy.....	73
181	7.1.2.1. Instantiating and Reseeding the Root RBGC Construction.....	73
182	7.1.2.2. Instantiating and Reseeding a Non-Root RBGC Construction	74
183	7.2. Conceptual Interfaces	74
184	7.2.1. RBGC Instantiation	74
185	7.2.1.1. Instantiation of the Root RBGC Construction	75
186	7.2.1.2. Instantiating an RBGC Construction Other Than the Root	79
187	7.2.2. Requesting the Generation of Pseudorandom Bits From an RBGC Construction.....	81
188	7.2.3. Reseeding an RBGC Construction.....	82
189	7.2.3.1. Reseed of the DRBG in the Root RBGC Construction	82
190	7.2.3.2. Reseed of the DRBG in an RBGC Construction Other Than the Root	84
191	7.3. RBGC Requirements.....	85
192	7.3.1. General RBGC Construction Requirements.....	85
193	7.3.2. Additional Requirements for the Root RBGC Construction	86
194	7.3.3. Additional Requirements for an RBGC Construction That is NOT the Root of a DRBG Chain....	87
195	8. Testing.....	88
196	8.1. Health Testing	88
197	8.1.1. Testing RBG Components	88
198	8.1.2. Handling Failures	88
199	8.1.2.1. Entropy-Source Failures.....	88
200	8.1.2.2. Failures by Non-Entropy-Source Components	89
201	8.2. Implementation Validation	89
202	References.....	91
203	Appendix A. Auxiliary Discussions (Informative)	94
204	A.1. Entropy vs. Security Strength	94
205	A.1.1. Entropy.....	94
206	A.1.2. Security Strength.....	94
207	A.1.3. A Side-by-Side Comparison	94
208	A.1.4. Entropy and Security Strength in This Recommendation	95
209	A.2. Generating Full-Entropy Output Using the RBG3(RS) Construction	96
210	A.3. Additional Considerations for RBGC Constructions.....	97

211	A.3.1. RBGC Tree Composition	98
212	A.3.2. Changes in the Tree Structure.....	100
213	A.3.3. Using Virtual Machines	100
214	A.3.4. Reseeding From Siblings of the Parent	102
215	Appendix B. RBG Examples (Informative).....	104
216	B.1. Direct DRBG Access in an RBG3 Construction	104
217	B.2. Example of an RBG1 Construction.....	105
218	B.2.1. Instantiation of the RBG1 Construction	106
219	B.2.2. Generation by the RBG1 Construction.....	108
220	B.3. Example Using Sub-DRBGs Based on an RBG1 Construction.....	109
221	B.3.1. Instantiation of the Sub-DRBGs.....	109
222	B.3.1.1. Instantiating Sub-DRBG1	110
223	B.3.1.2. Instantiating Sub-DRBG2	110
224	B.3.2. Pseudorandom Bit Generation by Sub-DRBGs.....	111
225	B.4. Example of an RBG2(P) Construction.....	111
226	B.4.1. Instantiation of an RBG2(P) Construction	112
227	B.4.2. Generation Using an RBG2(P) Construction.....	113
228	B.4.3. Reseeding an RBG2(P) Construction	114
229	B.5. Example of an RBG3(XOR) Construction.....	114
230	B.5.1. Instantiation of an RBG3(XOR) Construction	115
231	B.5.2. Generation by an RBG3(XOR) Construction.....	116
232	B.5.2.1. Generation.....	116
233	B.5.2.2. Get_conditioned_full-entropy_input Function	118
234	B.5.3. Reseeding an RBG3(XOR) Construction	119
235	B.6. Example of an RBG3(RS) Construction.....	119
236	B.6.1. Instantiation of an RBG3(RS) Construction	121
237	B.6.2. Generation by an RBG3(RS) Construction.....	122
238	B.6.3. Generation by the Directly Accessible DRBG	124
239	B.6.4. Reseeding a DRBG	125
240	B.7. DRBG Chains Using the RBGC Construction.....	125
241	B.7.1. Instantiation of the RBGC Constructors	126
242	B.7.1.1. Instantiation of the Root RBGC Construction.....	126
243	B.7.1.2. Instantiation of a Child RBGC Construction ($RBGC_2$)	127
244	B.7.2. Requesting the Generation of Pseudorandom Bits	128

245	B.7.3. Reseeding an RBGC Construction.....	129
246	B.7.3.1. Reseeding the Root RBGC Construction.....	129
247	B.7.3.2. Reseeding a Child RBGC Construction	130
248	Appendix C. Addendum to SP 800-90A: Instantiating and Reseeding a CTR_DRBG	131
249	C.1. Background and Scope.....	131
250	C.2. CTR_DRBG Without a Derivation Function.....	131
251	C.3. CTR_DRBG Using a Derivation Function	131
252	C.3.1. Derivation Keys and Constants.....	132
253	C.3.2. Derivation Function Using CMAC	132
254	C.3.3. Derivation Function Using CBC-MAC.....	132
255	Appendix D. List of Abbreviations and Acronyms	134
256	D.1. List of Symbols	134
257	Appendix E. Glossary.....	136
258	List of Tables	
259	Table 1. RBG capabilities	4
260	Table 2. Key lengths for the hash-based conditioning functions.....	27
261	Table 3. Highest security strength for the DRBG's cryptographic primitive	53
262	Table 4. Values for generating full-entropy bits by an RBG3(RS) construction	96
263	List of Figures	
264	Fig. 1. DRBG instantiations	8
265	Fig. 2. Example of an RBG security boundary within a cryptographic module	10
266	Fig. 3. General function calls.....	14
267	Fig. 4. DRBG_Instantiate function	15
268	Fig. 5. DRBG_Instantiate request	16
269	Fig. 6. DRBG_Generate function	17
270	Fig. 7. DRBG_Generate_request	18
271	Fig. 8. DRBG_Reseed function.....	18
272	Fig. 9. DRBG_Reseed_request.....	19
273	Fig. 10. Get_entropy_bitstring function	20
274	Fig. 11. RBG3 instantiate function.....	21
275	Fig. 12. RBG3(XOR) or RBG3(RS) instantiation request.....	22
276	Fig. 13. RBG3 generate functions	22

277	Fig. 14. Generic RBG3 generation process	23
278	Fig. 15. Generic structure of the RBG1 construction	34
279	Fig. 16. Instantiation using an RBG2(P) construction as a randomness source	36
280	Fig. 17. Instantiation using an RBG3(XOR) or RBG3(RS) construction as a randomness source	36
281	Fig. 18. RBG1 construction with sub-DRBGs	39
282	Fig. 19. Generic structure of the RBG2 construction	45
283	Fig. 20. RBG2 generate request following an optional reseed request.....	48
284	Fig. 21. Reseed request from an application.....	50
285	Fig. 22. Generic structure of the RBG3(XOR) construction.....	55
286	Fig. 23. Generic structure of the RBG3(RS) construction	61
287	Fig. 24. Sequence of RBG3(RS) generate requests.....	63
288	Fig. 25. Flow of the RBG3(RS)_Generate function.....	64
289	Fig. 26. Direct DRBG generate requests.....	65
290	Fig. 27. Modification of the DRBG_Reseed function	67
291	Fig. 28. Request extra bits before reseeding.....	68
292	Fig. 29. DRBG tree using the RBGC construction	72
293	Fig. 30. Instantiation of the DRBG in the root RBGC construction using an RBG2 or RBG3 construction as the randomness source.....	76
295	Fig. 31. Instantiation of the DRBG in the root RBGC construction using a full-entropy source as a randomness source.....	78
297	Fig. 32. Instantiation of the DRBG in RBGC_n using RBGC_{RS} as the randomness source.....	79
298	Fig. 33. Generate request received by the DRBG in an RBGC construction.....	81
299	Fig. 34. Reseed request received by the DRBG in the root RBGC construction	82
300	Fig. 35. Reseed request received by an RBGC construction other than the root	84
301	Fig. 36. Subtree in module B seeded by root RBGC of module A.....	98
302	Fig. 37. Subtree in module B seeded by a non-root DRBG of module A (i.e., DRBG₄)	99
303	Fig. 38. Subtree in module B seeded by DRBG₄ in module A.....	99
304	Fig. 39. Subtree in module B seeded by DRBG₂ of module A	100
305	Fig. 40. VM₁ and VM₂ with different virtual switches.....	101
306	Fig. 41. VM₁ and VM₂ with the same virtual switch but different port groups	101
307	Fig. 42. Acceptable external seeding for virtual machine RBGC constructions	101
308	Fig. 43. Acceptable external seeding for an RBGC construction in VM₂ but not in VM₁ and VM₃.....	102
309	Fig. 44. Application subtree obtaining reseed material from a sibling of its parent	103
310	Fig. 45. DRBG Instantiations	104
311	Fig. 46. Example of an RBG1 construction	106

312	Fig. 47. Sub-DRBGs based on an RBG1 construction	109
313	Fig. 48. Example of an RBG2 construction	112
314	Fig. 49. Example of an RBG3(XOR) construction	115
315	Fig. 50. Example of an RBG3(RS) construction	120
316	Fig. 51. Example of DRBG chains.....	126
317		

318 **Acknowledgments**

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

325 **1. Introduction and Purpose**

326 Cryptography and security applications make extensive use of random bits. However, the
327 generation of random bits is challenging in many practical applications of cryptography. The
328 National Institute of Standards and Technology (NIST) developed the Special Publication (SP) 800-
329 90 series to support the generation of high-quality random bits for both cryptographic and non-
330 cryptographic purposes. The SP 800-90 series consists of three parts:

- 331 1. SP 800-90A, *Recommendation for Random Number Generation Using Deterministic*
332 *Random Bit Generators*, specifies several **approved** deterministic random bit generator
333 (DRBG) mechanisms based on **approved** cryptographic algorithms that — once provided
334 with seed material that contains sufficient randomness — can be used to generate
335 random bits suitable for cryptographic applications.
- 336 2. SP 800-90B, *Recommendation for the Entropy Sources Used for Random Bit Generation*,
337 provides guidance for the development and validation of entropy sources, which are
338 mechanisms that generate entropy from physical or non-physical noise sources and that
339 can be used to generate the input for the seed material needed by a DRBG or for input to
340 an RBG.
- 341 3. SP 800-90C, *Recommendation for Random Bit Generator (RBG) Constructions*, specifies
342 constructions for random bit generators (RBGs) using 1) randomness sources (either
343 entropy sources that comply with SP 800-90B or RBGs that comply with SP 800-90C) and
344 2) DRBGs that comply with SP 800-90A. Four classes of RBGs are specified in this
345 document (see Sec. 4–7). SP 800-90C also provides high-level guidance for testing RBGs
346 for conformance to this recommendation.

347 Throughout this document, the phrase “this recommendation” refers to the aggregate of SP 800-
348 90A, SP 800-90B, and SP 800-90C, while the phrase “this document” refers only to SP 800-90C.

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

354 SP 800-90C has been developed in coordination with NIST’s Cryptographic Algorithm Validation
355 Program (CAVP) and Cryptographic Module Validation Program (CMVP). The document uses
356 “**shall**” and “**must**” to indicate requirements and uses “**should**” to indicate an important
357 recommendation. The term “**shall**” is used when a requirement is testable by a testing lab during
358 implementation validation using operational tests or a code review. The term “**must**” is used for
359 requirements that may not be testable by the CAVP or CMVP. An example of such a requirement
360 is one that demands certain actions and/or considerations from a system administrator. Meeting
361 these requirements can be verified by a CMVP review of the cryptographic module’s
362 documentation. If the requirement is determined to be testable at a later time (e.g., after SP 800-

363 90C is published and before it is revised), the CMVP will so indicate in the *Implementation*
364 *Guidance for FIPS 140-3 and the Cryptographic Module Validation Program* [FIPS_140IG].

365 **1.1. Audience**

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

370 **1.2. Document Organization**

371 This document is organized as follows:

- 372 • Section 2 provides background and preliminary information for understanding the
373 remainder of the document.
- 374 • Section 3 provides guidance on accessing and handling entropy sources, including the
375 external conditioning of entropy-source output to reduce bias and obtain full entropy
376 when needed.
- 377 • Sections 4, 5, 6, and 7 specify the RBG constructions, namely the RBG1, RBG2, RBG3, and
378 RBGC constructions, respectively.
- 379 • Section 8 discusses health and implementation validation testing.
- 380 • The References contain a list of papers and publications cited in this document.

381 The following informational appendices are also provided:

- 382 • Appendix A provides discussions on entropy versus security strength, generating output
383 using the RBG3(RS) construction, and computing platforms, as required by DRBG chains
384 using the RBGC construction.
- 385 • Appendix B provides examples of each RBG construction.
- 386 • Appendix C is an addendum for SP 800-90A that includes two additional derivation
387 functions that may be used with the CTR_DRBG. These functions will be moved into SP
388 800-90A as part of the next revision of that document.
- 389 • Appendix D provides a list of abbreviations, symbols, functions, and notations used in this
390 document.
- 391 • Appendix E provides a glossary with definitions for terms used in this document.

392 **2. General Information**393 **2.1. RBG Security**

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

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

404
$$1/2 + 2^{w-s-1} + \varepsilon,$$

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

407 An RBG that has been designed to support a security strength of s bits is suitable for any
408 application with a targeted security strength that does not exceed s . An RBG that is compliant
409 with this recommendation can support requests for output with a security strength of 128, 192,
410 or 256 bits, except for an RBG3 construction (as described in Sec. 6), which can provide full-
411 entropy output.¹

412 A bitstring with full entropy has an amount of entropy equal to its length. Full-entropy bitstrings
413 are important for cryptographic applications, as these bitstrings have ideal randomness
414 properties and may be used for any cryptographic purpose. They may be truncated to any length
415 such that the amount of entropy in the truncated bitstring is equal to its length. However, due to
416 the difficulty of generating and testing full-entropy bitstrings, this recommendation assumes that
417 a bitstring has full entropy if the amount of entropy per bit is at least $1 - \varepsilon$, where ε is at most
418 2^{-32} . NIST Internal Report (IR) 8427 [NISTIR_8427] provides a justification for the selection of ε .

419

420 **2.2. RBG Constructions**

421 A *construction* is a method of designing an RBG to accomplish a specific goal. Four classes of RBG
422 constructions are defined in this document: RBG1, RBG2, RBG3, and RBGC (see Table 1). Each
423 RBG includes a DRBG from SP 800-90A and is based on the use of a randomness source that is
424 validated for compliance with SP 800-90B or SP 800-90C. Once instantiated, a DRBG can generate
425 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.

426

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

427 In Table 1:

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

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

442 An RBG2 construction includes one or more entropy sources that are used to instantiate the
 443 DRBG and may (optionally) be used for reseeding if a reseed capability is implemented. Prediction
 444 resistance may be provided to the RBG output when reseeding is performed. The construction

445 has two variants: an RBG2(P) construction uses a physical entropy source to provide entropy,
446 while an RBG2(NP) construction uses a non-physical entropy source. An RBG2 construction
447 cannot provide full-entropy output (see Sec. 5).

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

451 This construction has two types:

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

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

464 This document also provides procedures for acquiring entropy from an entropy source and
465 conditioning the output to provide a bitstring with full entropy (see Sec. 3.2). SP 800-90A provides
466 constructions for instantiating and reseeding DRBGs and requesting the generation of
467 pseudorandom bitstrings.

468 All constructions in SP 800-90C are described in pseudocode as well as text. The pseudocode
469 conventions are not intended to constrain real-world implementations but to provide a
470 consistent notation to describe the constructions.

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

474 By convention and unless otherwise specified, integers are unsigned 32-bit values. When used as
475 bitstrings, they are represented in the big-endian format.

476 **2.3. Sources of Randomness for an RBG**

477 The RBG constructions specified in this document are based on the use of validated entropy
478 sources — mechanisms that provide entropy for an RBG. Some RBG constructions access these
479 entropy sources directly to obtain entropy. Other constructions fulfill their entropy requirements
480 by accessing another RBG as a randomness source, in which case the RBG used as a randomness
481 source may include an entropy source or have a predecessor that includes an entropy source.

482 SP 800-90B provides guidance for the development and validation of entropy sources. Validated
483 entropy sources (i.e., entropy sources that have been successfully validated by the CMVP as
484 complying with SP 800-90B) reliably provide fixed-length outputs and a specified minimum
485 amount of entropy for each output (e.g., each 8-bit output has been validated as providing at
486 least five bits of entropy).²

487 An entropy source is a *physical entropy source* if the primary noise source within the entropy
488 source is physical — that is, it uses a dedicated hardware design to provide entropy (e.g., from
489 ring oscillators, thermal noise, shot noise, jitter, or metastability). Similarly, a validated entropy
490 source is a *non-physical entropy source* if the primary noise source within the entropy source is
491 non-physical — that is, entropy is provided by system data (e.g., system time or the entropy
492 present in the RAM data) or human interaction (e.g., mouse movements). The entropy source
493 type (i.e., physical or non-physical) is certified during SP 800-90B validation.

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

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

502 **Method 1:** The RBG implementation includes one or more independent, validated physical
503 entropy sources; one or more validated non-physical entropy sources may also be included
504 in the implementation. Only the entropy in a bitstring that is provided from physical entropy
505 sources is counted toward fulfilling the amount of entropy requested in an entropy request.
506 Any entropy in a bitstring that is provided by a non-physical entropy source is not counted,
507 even if bitstrings produced by the non-physical entropy source are included in the
508 concatenated bitstring that is used by the RBG.

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

513 *Example:* Let pes_i be the i^{th} output of a physical entropy source and $npes_j$ be the j^{th} output of
514 a non-physical entropy source. If an implementation consists of one physical and one non-
515 physical entropy source, and a request has been made for 128 bits of entropy, the
516 concatenated bitstring might be something like:

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

518 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 precede “entropy sources.” The use of the term “validated entropy source” may be shortened to just “entropy source” to avoid repetition.

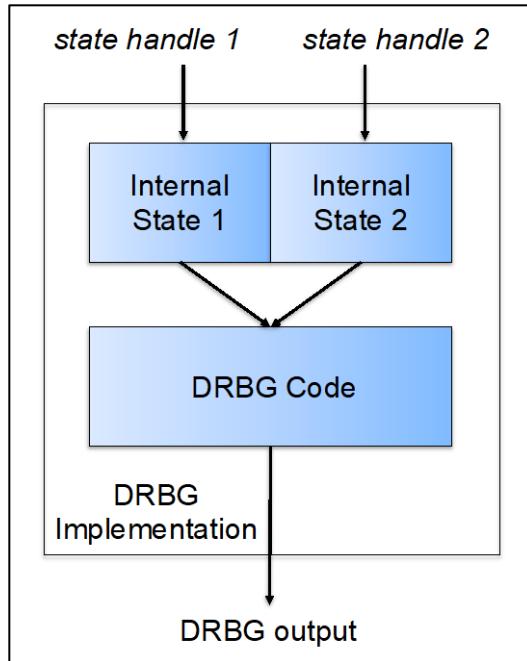
519 According to Method 1, only the entropy in $pes_1, pes_2, \dots, pes_n$ would be counted toward
520 fulfilling the 128-bit entropy request. Any entropy in $npes_1, \dots, npes_m$ is not counted.
521 According to Method 2, all the entropy in $pes_1, pes_2, \dots, pes_n$ and in $npes_1, npes_2, \dots, npes_m$ is
522 counted.
523 When multiple entropy sources are used, there is no requirement on the order in which the
524 entropy sources are accessed or the number of times that each entropy source is accessed to
525 fulfill an entropy request. For example, if two physical entropy sources are used, it is possible
526 that a request would be fulfilled by only one of the entropy sources because entropy is not
527 available at the time of the request from the other entropy source. However, the Method 1 or
528 Method 2 criteria for counting entropy still apply, providing that the entropy sources are
529 independent.
530 This recommendation assumes that the entropy produced by a validated physical entropy source
531 is generally more reliable than the entropy produced by a validated non-physical entropy source
532 since non-physical entropy sources are typically influenced by human actions or network events,
533 the unpredictability of which is difficult to accurately quantify. Therefore, Method 1 is considered
534 to provide more assurance that the concatenated bitstring contains at least the requested
535 amount of entropy (e.g., 128 bits for a 128-bit AES key). Note that the RBG2(P) and RBG3
536 constructions only count entropy using Method 1 (see Sec. 5 and 6, respectively).

537 **2.4. DRBGs**

538 Approved DRBGs are specified in SP 800-90A. A DRBG includes instantiate, generate, and health-
539 testing functions and may also include reseed and uninstantiate functions. The instantiation of a
540 DRBG involves acquiring sufficient randomness to initialize the DRBG to support a targeted
541 security strength and establish the internal state, which includes the secret information for
542 operating the DRBG. The generate function produces output upon request and updates the
543 internal state. Health testing is used to determine that the DRBG continues to operate correctly.
544 Reseeding introduces fresh randomness into the DRBG's internal state and is used to recover
545 from a potential (or actual) compromise (see Sec. 2.4.2 for an additional discussion). An
546 uninstantiate function is used to terminate a DRBG instantiation and destroy the information in
547 its internal state.

548 **2.4.1. DRBG Instantiations**

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



556

Fig. 1. DRBG instantiations

558 Each DRBG instantiation is initialized with input from some randomness source that establishes
559 the security strength(s) that can be supported by the DRBG. During this process, an optional but
560 recommended personalization string may also be used to differentiate between instantiations in
561 addition to the output of the randomness source. The personalization string could, for example,
562 include information particular to the instantiation or contain entropy collected during system
563 activity (e.g., from a non-validated entropy source). An implementation **should** allow the use of
564 a personalization string. More information on personalization strings is provided in SP 800-90A.

565 A DRBG may be implemented to accept additional input during operation from the randomness
566 source (e.g., to reseed the DRBG) and/or additional input from inside or outside of the
567 cryptographic module that contains the DRBG. This additional input could, for example, include
568 information particular to a request for generation or reseeding or could contain entropy collected
569 during system activity (e.g., from a validated or non-validated entropy source).³ A capability to
570 handle additional input is recommended for an implementation.

571 **2.4.2. Reseeding, Prediction Resistance, and Compromise Recovery**

572 Under some circumstances, the internal state of an RBG (containing the RBG's secret
573 information) could be leaked to an adversary. This might happen as the result of a side-channel
574 attack or a serious compromise of the computer on which the DRBG runs and may not be
575 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.

576 In order to limit damage due to a compromised state, all DRBGs in SP 800-90A are designed with
577 *backtracking resistance* — that is, learning the DRBG’s current internal state does not provide
578 knowledge of previous outputs. Since all RBGs in SP 800-90C are based on the use of the DRBGs
579 in SP 800-90A, the RBGs specified in this document also inherit this property.

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

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

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

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

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

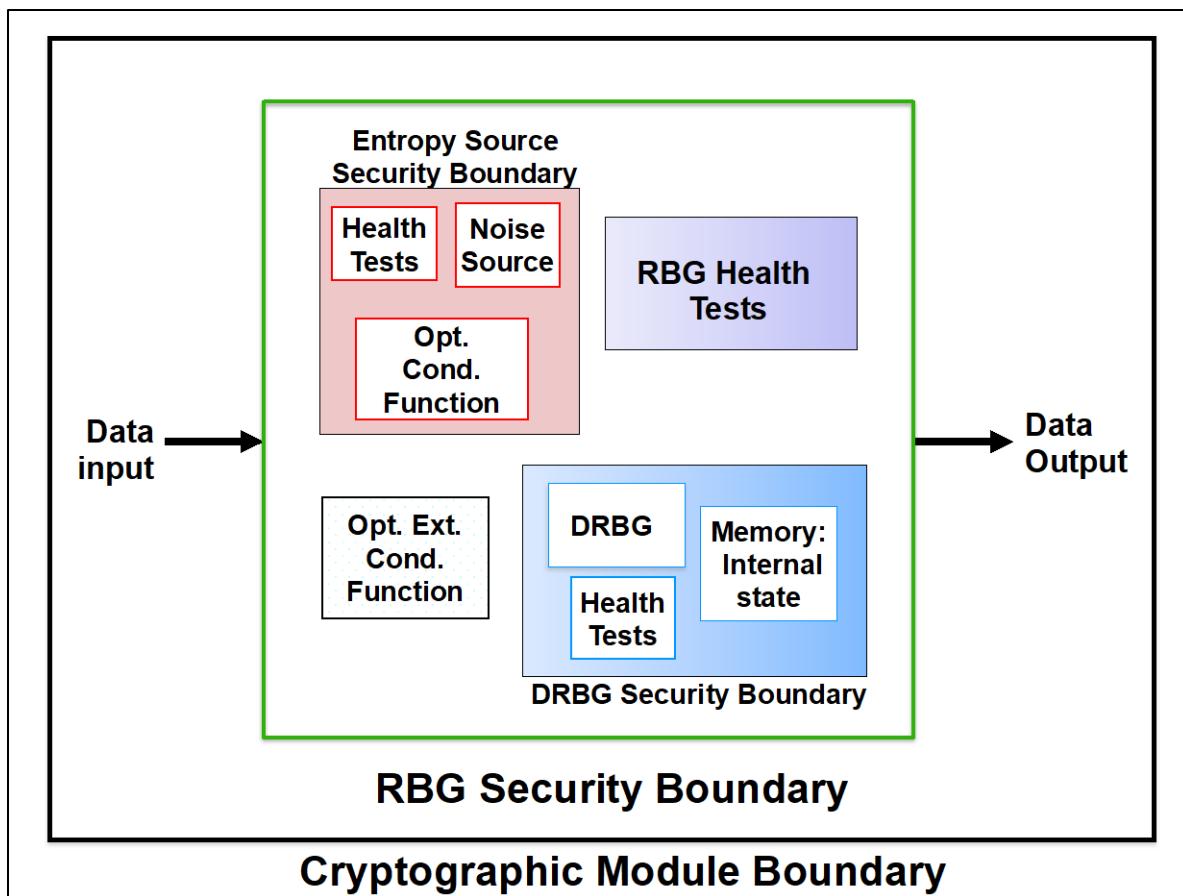
606 The RBG1, RBG2, and RBGC constructions provide output with a security strength that depends
607 on the security strength of the DRBG instantiation within the RBG and the length of the output.
608 These constructions do not provide output with full entropy and **must not** be used by applications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

611 Although reseeding provides fresh randomness that is incorporated into an already instantiated
612 DRBG at a security strength of s bits, the reseed process does not increase the DRBG’s security
613 strength. For example, a reseed of a DRBG that has been instantiated to support a security
614 strength of 128 bits does not increase the DRBG’s security strength to 256 bits when reseeding
615 with 128 bits of fresh entropy.

616 **2.5. RBG Security Boundaries**

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

621 The primary components of an RBG are a randomness source, a DRBG, and health tests for the
622 RBG. RBG input (e.g., entropy bits and a personalization string) **shall** enter an RBG only as
623 specified in the functions described in Sec. 2.8. The security boundary of a DRBG is discussed in
624 SP 800-90A, and the security boundary for an entropy source is discussed in SP 800-90B. Both the
625 entropy source and the DRBG contain their own health tests within their respective security
626 boundaries.



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

628 Figure 2 shows an example RBG implemented within a FIPS 140-validated cryptographic module.
629 In this figure, the RBG security boundary is completely contained within the cryptographic
630 module boundary. The data input may be a personalization string or additional input (see Sec.
631 2.4.1). The data output is status information and possibly random bits or a state handle. Within
632 the RBG security boundary of the figure are an entropy source and a DRBG, each with its own
633

634 conceptual security boundary. An entropy-source security boundary includes a noise source,
635 health tests, and (optionally) a conditioning component. A DRBG security boundary contains the
636 chosen DRBG, memory for the internal state, and health tests. An RBG security boundary contains
637 health tests and an (optional) external conditioning function. The RBG2 and RBG3 constructions
638 in Sec. 5 and 6, respectively, use this model.

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

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

646 2.6. Assumptions and Assertions

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

- 649 1. An entropy source is independent of another entropy source if their security boundaries
650 do not overlap (e.g., they reside in separate cryptographic modules, or one is a physical
651 entropy source and the other is a non-physical entropy source).
- 652 2. Entropy sources that have been validated for conformance to SP 800-90B are used to
653 provide seed material for seeding and reseeding a DRBG or providing entropy for an RBG3
654 construction. The output of non-validated entropy sources is only used as additional
655 input.

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

- 658 3. An entropy source outputs no more than 2^{64} bits. The number of output bits from the
659 RBG is at most 2^{64} bits for a DRBG instantiation. In the case of an RBG1 construction with
660 one or more subordinate DRBGs, the output limit applies to the total output provided by
661 the RBG1 construction and its subordinate DRBGs.
- 662 4. Each entropy-source output has a fixed length ES_len (in bits).
- 663 5. Each entropy-source output is assumed to contain a fixed amount of entropy, denoted as
664 $ES_entropy$, that was assessed during entropy-source implementation validation. See SP
665 800-90B for entropy estimation.
- 666 6. Each entropy source has been characterized as either a physical entropy source or a non-
667 physical entropy source upon successful validation.
- 668 7. The outputs from a single entropy source can be concatenated. The entropy of the
669 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:

- 707 • For an RBG1 construction, a bitstring at least $3s/2$ bits long is needed from a
708 randomness source (an RBG) providing at least s bits of security strength (see Sec.
709 4).
710 • For an RBG2 or RBG3 construction, bitstrings with at least $3s/2$ bits of entropy are
711 needed from the entropy source(s) (see Sec. 5 and 6, respectively).
712 • For an RBGC construction that is the root of a tree of RBGC constructions, at least
713 $3s/2$ bits of entropy are needed from the randomness source when the initial
714 randomness source is a full-entropy source or RBG3 construction. If the initial
715 randomness source is an RBG2 construction, a bitstring at least $3s/2$ bits long is
716 needed from the randomness source (see Sec. 7).
717 • For an RBGC construction that is not the root of the tree, a bitstring at least $3s/2$
718 bits long is needed from the construction's randomness source (see Sec. 7).
- 719 13. One or more of the constructions provided herein are used in the design of an RBG.
720 14. All components of an RBG2 and RBG3 construction (as specified in Sec. 5 and 6) reside
721 within the physical boundary of a single FIPS 140-validated cryptographic module.
722 15. All RBGC constructions in a DRBG chain reside on the same computing platform.
723 16. The DRBGs specified in SP 800-90A are assumed to meet their explicit security claims (e.g.,
724 backtracking resistance, claimed security strength, etc.).
725 17. A sub-DRBG is considered to be part of the RBG1 construction that initializes it.
726 18. The RBG1 construction and its sub-DRBGs reside within the physical boundary of a single
727 FIPS 140-validated cryptographic module.

728 **2.7. General Implementation and Use Requirements and Recommendations**

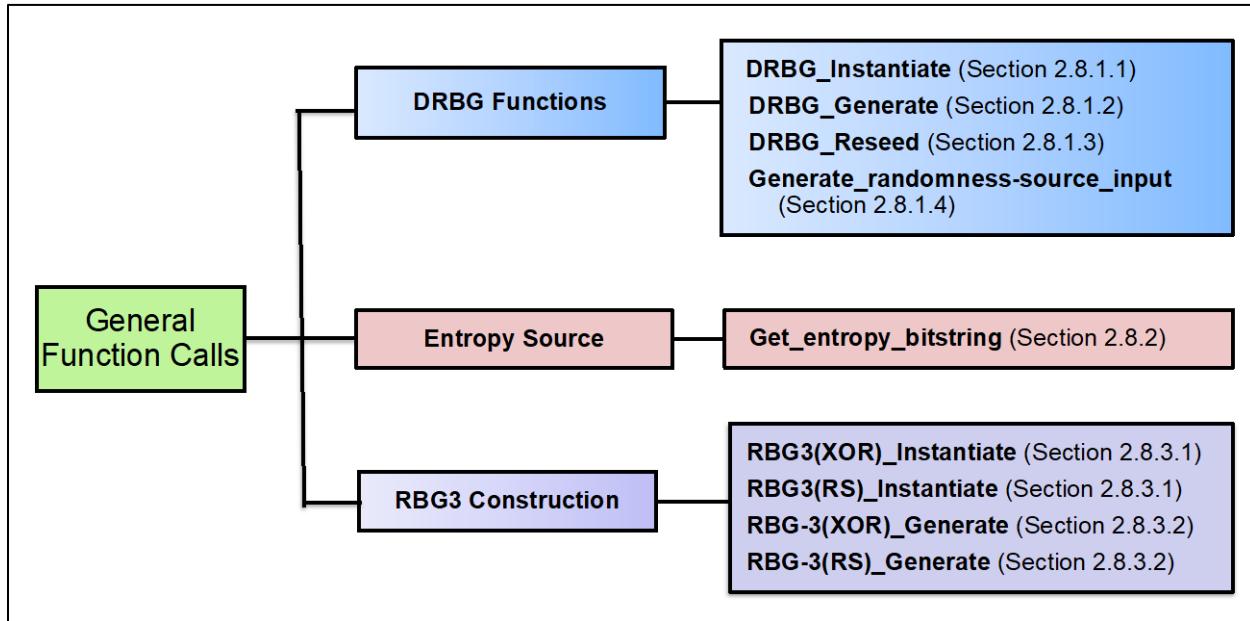
729 When implementing the RBG constructions specified in this recommendation, an
730 implementation:

- 731 1. **Shall** destroy intermediate values before exiting the function or routine in which they are
732 used,
- 733 2. **Shall** employ an “atomic” generate operation whereby a generate request is completed
734 before using any of the requested bits, and
- 735 3. **Should** be implemented with the capability to support a security strength of 256 bits or
736 to provide full-entropy output.

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

742 **2.8. General Function Calls**

743 Functions used within this document for accessing the DRBGs in SP 800-90A, the entropy sources
744 in SP 800-90B, and the RBG3 constructions specified in SP 800-90C are provided below and in Fig.
745 3.



746

Fig. 3. General function calls

747
748 Each function returns a status code that **must** be checked (e.g., a status of success or failure by
749 the function).

- 750 • If the status code indicates a success, then additional information may also be returned,
751 such as a state handle from an instantiate function or the bits that were requested to be
752 generated during a generate function.
- 753 • If the status code indicates a failure of an RBG component, then see item 10 in Sec. 2.6
754 and Sec. 8.1.2 for error-handling guidance. Note that if the status code does not indicate
755 a success, an invalid output (e.g., a null bitstring) **shall** be returned with the status code if
756 information other than the status code could be returned.

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

763 **2.8.1. DRBG Functions**

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

768 At least two functions are required in a DRBG:

- 769 1. An instantiate function that seeds the DRBG using the output of a randomness source and
770 other optional input (see Sec. 2.8.1.1) and
- 771 2. A generate function that produces output for use by a consuming application (see Sec.
772 2.8.1.2).

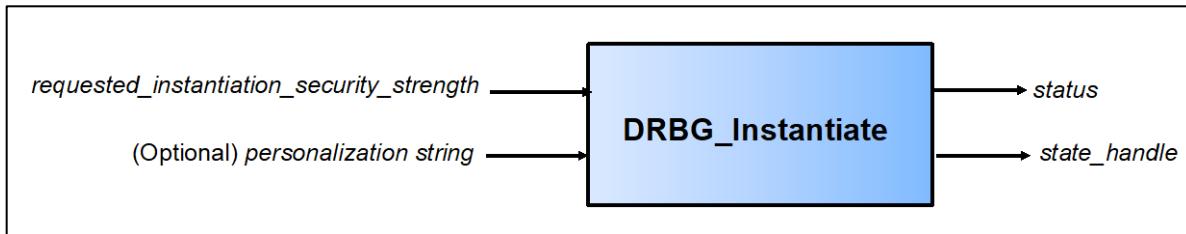
773 A DRBG may also support a reseed function (see Sec. 2.8.1.3).

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

778 The use of the **DRBG_Uninstantiate** function

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

781 $(status, state_handle) = \text{DRBG_Instantiate} (requested_instantiation_security_strength,$
782 $personalization_string)$.



783

784 **Fig. 4. DRBG_Instantiate function**

785 The **DRBG_Instantiate** function (shown in Fig. 4) is used to instantiate a DRBG at the
786 *requested_instantiation_security_strength* using the output of a randomness source⁴ and an
787 optional *personalization_string* to create a seed. As stated in Sec. 2.4.1, a *personalization_string*
788 is optional but strongly recommended. Details about the **DRBG_Instantiate** function are
789 provided in SP 800-90A.

790 If the *status* code returned for the **DRBG_Instantiate** function indicates a success (i.e., the DRBG
791 has been instantiated at the requested security strength), a state handle may⁵ 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.

792 indicate the particular DRBG instance (i.e., pointing to the internal state to be used by this
 793 instance). When provided by the **DRBG_Instantiate** function, the state handle is used in
 794 subsequent calls to the DRBG (e.g., during a **DRBG_Generate** call) to reference the internal state
 795 information for the instantiation. The information in the internal state includes the security
 796 strength of the instantiation and other information that changes during DRBG execution (see SP
 797 800-90A for each DRBG design).

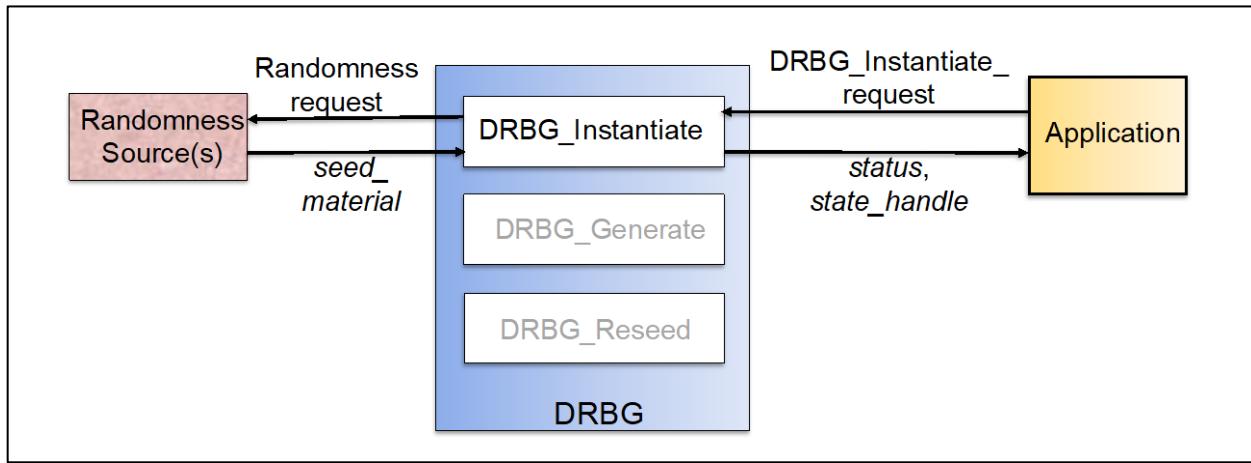
798 When the DRBG has been instantiated at the requested security strength, the DRBG will operate
 799 at that security strength even if the security strength requested in subsequent **DRBG_Generate**
 800 calls (see Sec. 2.8.1.2) is less than the instantiated security strength. For example, if a DRBG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

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

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

807 $(status, state_handle) = \text{DRBG_Instantiate_request}(requested_instantiation_security_strength,$
 808 $personalization_string)$.

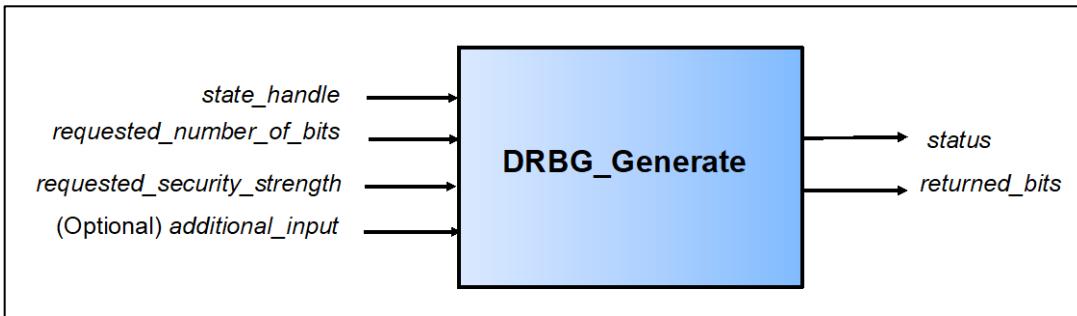
809 As shown in Fig. 5, a **DRBG_Instantiate_request** received by a DRBG results in the execution of
 810 the DRBG's instantiate function, providing the input parameters for that function. The
 811 **DRBG_Instantiate** function then obtains *seed_material* from the randomness source(s),
 812 instantiates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



814
 815 Fig. 5. DRBG_Instantiate request

816 **2.8.1.1. DRBG Generation Request**

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

818 $(status, returned_bits) = \text{DRBG_Generate}(\text{state_handle}, \text{requested_number_of_bits},$
819 $\text{requested_security_strength}, \text{(Optional) additional_input}).$ 

820

821 **Fig. 6. DRBG_Generate function**822 The **DRBG_Generate** function (shown in Fig. 6) is used to generate a specified number of bits.
823 If a suitable *state_handle* is available, it is included as input to indicate the DRBG instance to be
824 used. The number of bits to be returned and the security strength that the DRBG needs to support
825 for generating the bitstring are provided with (optional) additional input. As stated in Sec. 2.4.1,
826 the ability to accept additional input is recommended.827 The **DRBG_Generate** function returns status information — either an indication of success or
828 an error. If the returned status code indicates a success, the requested bits are returned.

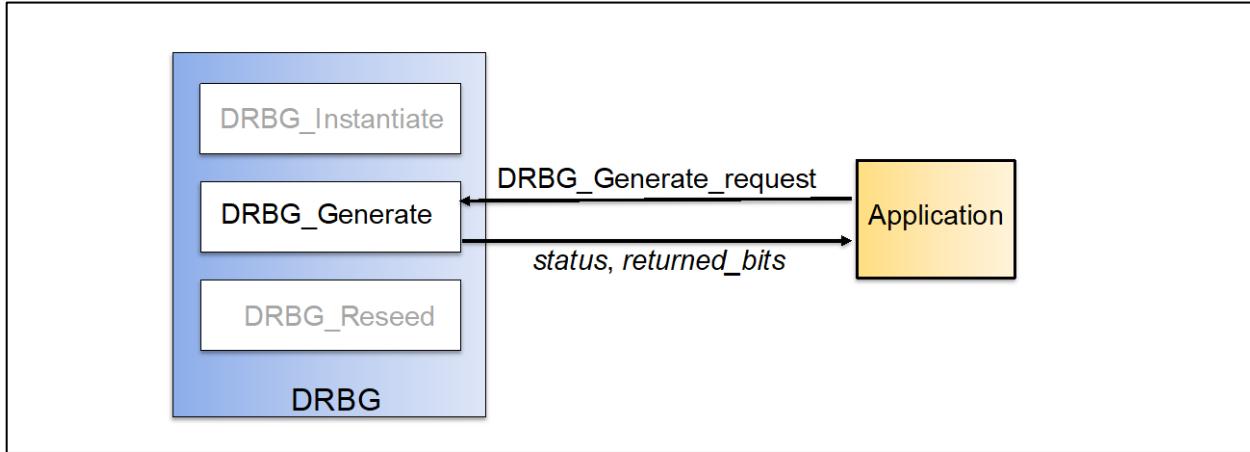
- 829 • If
- requested_number_of_bits*
- is equal to or greater than the instantiated security strength,
-
- 830 the security strength that the
- returned_bits*
- can support (if used as a key) is:

831 $ss_key = \text{the instantiated security strength},$ 832 where *ss_key* is the security strength of the key.

- 833 • If the
- requested_number_of_bits*
- is less than the instantiated security strength, and the
-
- 834
- returned_bits*
- are to be used as a key, the key is capable of supporting a security strength
-
- 835 of:

836 $ss_key = \text{requested_number_of_bits}.$ 837 If the status code indicates an error, the *returned_bits* consists of a *Null* bitstring. An example of
838 a condition in which an error indication may be returned includes a request for a security strength
839 that exceeds the instantiated security strength for the DRBG.840 Details about the **DRBG_Generate** function are provided in SP 800-90A.841 The **DRBG_Generate** function is requested by an application using a
842 **DRBG_Generate_request**:843 $(status, returned_bits) = \text{DRBG_Generate_request}(\text{state_handle}, \text{requested_number_of_bits},$
844 $\text{requested_security_strength}, \text{additional_input}).$

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

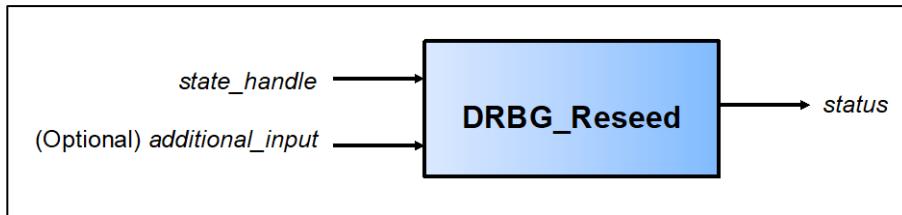


849

850 **Fig. 7. DRBG_Generate_request**851 **2.8.1.2. DRBG Reseed**

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

855 $status = \text{DRBG_Reseed}(\text{state_handle}, \text{additional_input})$.



856

857 **Fig. 8. DRBG_Reseed function**

858 A **DRBG_Reseed** function (shown in Fig. 8) is used to acquire at least s bits of fresh randomness
 859 for the DRBG instance indicated by the state handle (or the only instance if no state handle has
 860 been provided), where s is the security strength of the DRBG to be reseeded.⁷ In addition to the
 861 seed material provided from the DRBG's randomness source(s) during reseeding, optional
 862 *additional_input* may be incorporated into the reseed process. As discussed in Sec. 2.4.1, the
 863 capability for handling and using additional input is recommended. Details about the
 864 **DRBG_Reseed** function are provided in SP 800-90A.

⁶ Note that this does not increase the security strength of the DRBG.

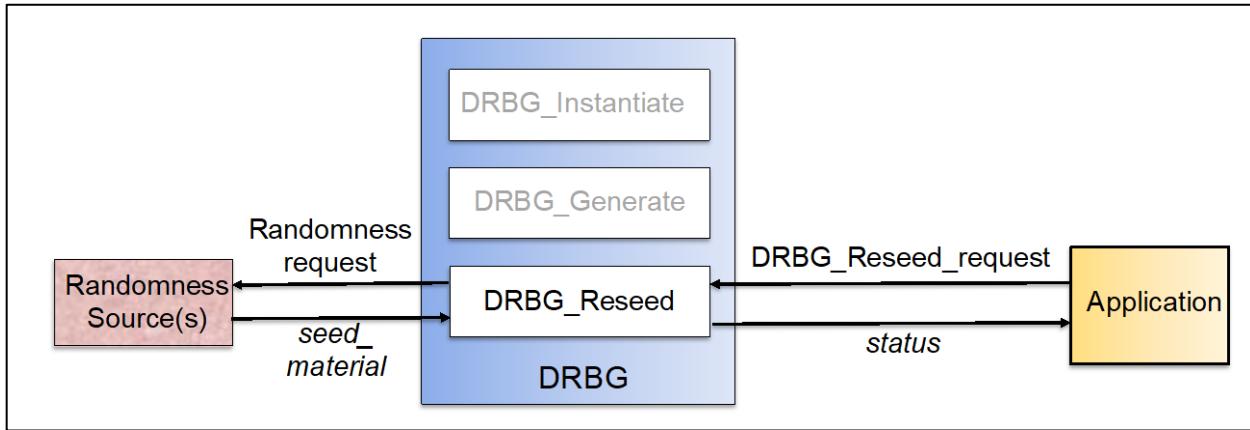
⁷ The value of s may be available in the DRBG's internal state (see SP 800-90A).

865 An indication of the *status* is returned.

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

867 $status = \text{DRBG_Reseed_request}(\text{state_handle}, \text{additional_input})$.

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



872

873 Fig. 9. DRBG_Reseed_request

874 2.8.1.3. Get_randomness-source_input Call

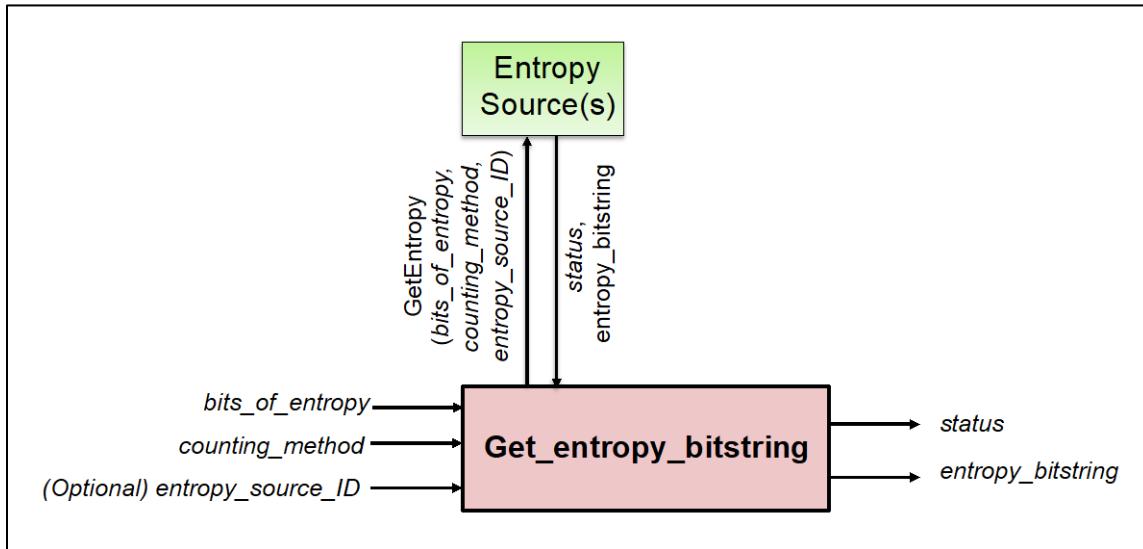
875 In SP 800-90A, a **Get_randomness-source_input** call is used in the **DRBG_Instantiate** function
 876 and **DRBG_Reseed** function to indicate when a randomness source needs to be accessed to
 877 obtain seed material. Details are not provided in SP 800-90A about how the **Get_randomness-**
 878 **source_input** call needs to be implemented. SP 800-90C provides guidance on how the call
 879 **should** be implemented based on various situations (e.g., the randomness source and the RBG
 880 construction used). Sections 3.2.2, 4, 5, 6, and 7 provide instructions for obtaining input from a
 881 randomness source when the **Get_randomness-source_input** call is encountered in SP 800-90A.

882 2.8.2. Interfacing With Entropy Sources

883 A single entropy source request may not be sufficient to obtain the entropy required for seeding
 884 and reseeding a DRBG and for providing input for the exclusive-or operation in an RBG3(XOR)
 885 construction (see Sec. 6.4.1). SP 800-90C uses the term **Get_entropy_bitstring** to identify the
 886 process of obtaining the required entropy from one or more entropy sources. For convenience
 887 in describing the RBG constructions, this process is represented as a function whose input
 888 includes an indication of the amount of entropy that is needed from the entropy source(s) and
 889 whose output includes a status report on the success or failure of the process. If the process is
 890 successful, a bitstring containing the requested entropy is produced (see Fig. 10). The
 891 **Get_entropy_bitstring** function is invoked herein as:

892 $(status, entropy_bitstring) = \text{Get_entropy_bitstring}(bits_of_entropy, counting_method,$
 893 $entropy_source_ID),$

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



898

899 **Fig. 10. Get_entropy_bitstring function**

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

904 **2.8.3. Interfacing With an RBG3 Construction**

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

908 **2.8.3.1. Instantiating a DRBG Within an RBG3 Construction**

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

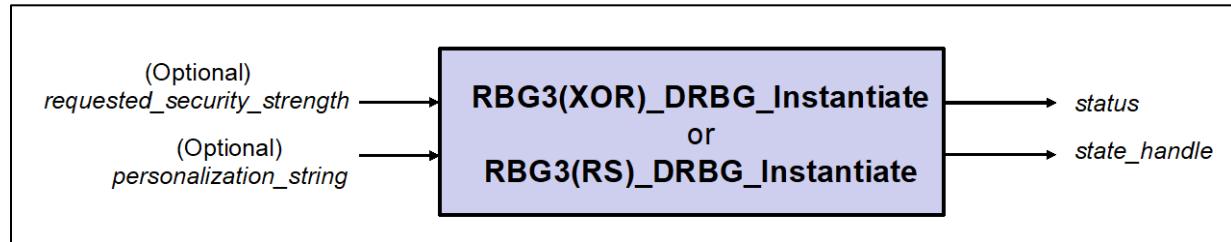
910 $(status, state_handle) = \text{RBG3(XOR)}\text{-Instantiate}(requested_security_strength,$

911 $personalization_string)$

912 and

913 $(status, state_handle) = \text{RBG3(RS)}\text{-Instantiate}(requested_security_strength,$

914 $personalization_string).$



915

916

Fig. 11. RBG3 instantiate function

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

924 If the returned status code indicates a success, a state handle may be returned to indicate the
 925 DRBG instance that is to be used by the construction (i.e., the state handle points to the internal
 926 state used by this instance of the DRBG within the RBG3 construction). If multiple instances of
 927 the DRBG are used (in addition to the DRBG instance used by the RBG3 construction), a separate
 928 state handle is returned for each instance. When provided, the state handle is used in subsequent
 929 calls to that RBG (e.g., during a call to the RBG3 generate function; see Sec. 2.8.3.2) or when
 930 accessing the DRBG directly (e.g., during a reseed of the DRBG; see Sec. 6.4.1.4). If the status
 931 code indicates an error (e.g., entropy is not currently available, or the entropy source has failed),
 932 an invalid (e.g., *Null*) state handle is returned.

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

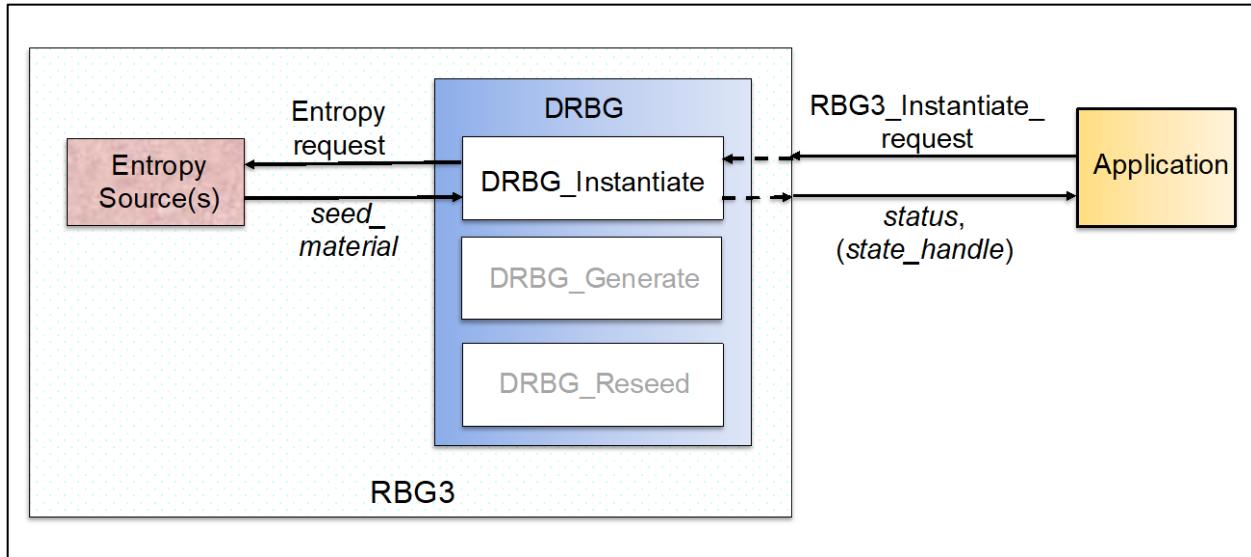
935 $(status, state_handle) = \text{Instantiate_RBG3_DRBG_request}(requested_security_strength,$
 936 $personalization_string)$.

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

941 The security strength of the DRBG within an RBG3 construction is the highest security strength
 942 that can be supported by the DRBG design (see Sec. 6). The *requested_security_strength*
 943 parameter in the **Instantiate_RBG3_DRBG_request** should be interpreted (in the case of the
 944 RBG3 construction) as the minimum security strength that is required by the consuming
 945 application if entropy-source failures are undetected. Therefore, if the
 946 *requested_security_strength* parameter is provided as input, it is compared against the value of
 947 the highest security strength that can be supported by the DRBG. If the
 948 *requested_security_strength* exceeds the security strength that can be supported by the DRBG,

949 then an error indication is returned as the *status* in response to the
 950 **Instantiate_RBG3_DRBG_request**.

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



954

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

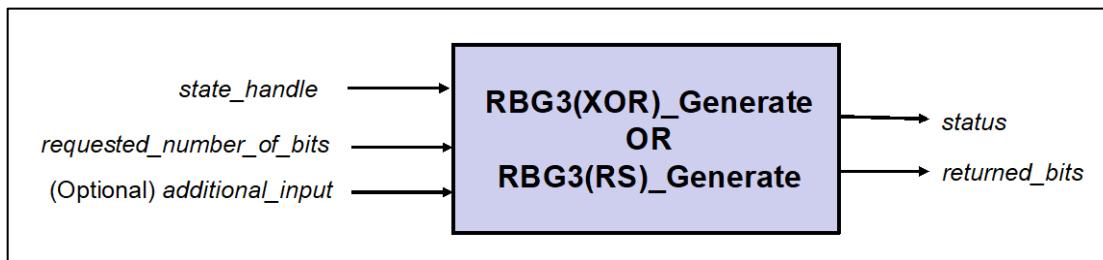
956 2.8.3.2. Generation Using an RBG3 Construction

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

960 $(status, returned_bits) = \text{RBG3(XOR)}_\text{Generate}(state_handle, requested_number_of_bits,$
 961 $additional_input)$

962 and

963 $(status, returned_bits) = \text{RBG3(RS)}_\text{Generate}(state_handle,$
 964 $requested_number_of_bits, additional_input).$



965

966

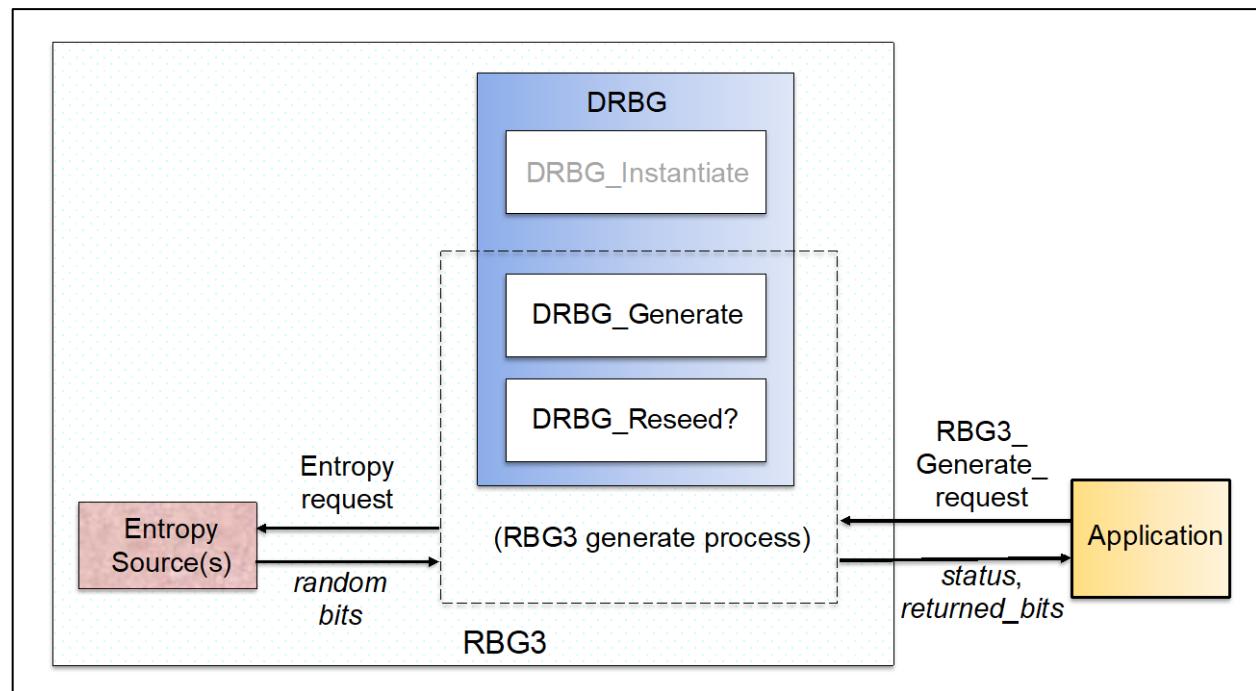
Fig. 13. RBG3 generate functions

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

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

974 $(status, returned_bits) = \text{RBG3_Generate_request}(\text{state_handle}, \text{requested_number_of_bits},$
975 $\text{requested_security_strength}, \text{additional_input})$.

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



984
985 Fig. 14. Generic RBG3 generation process
986

987 **3. Accessing Entropy Source Output**

988 The security provided by an RBG is based on the use of validated entropy sources. Section 3.1
989 discusses the use of the **Get_entropy_bitstring** process to request entropy from one or more
990 entropy sources. Section 3.2 discusses the conditioning of the output of one or more entropy
991 sources before further use by an RBG.

992 **3.1. Get_entropy_bitstring Process**

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

998 In many cases, the **Get_entropy_bitstring** process will need to query an entropy source (or a set
999 of entropy sources) multiple times to obtain the amount of entropy requested. The details of the
1000 process are not specified in this document but are left to the developer to implement
1001 appropriately for the selected entropy source(s). However, the following behavior of the
1002 **Get_entropy_bitstring** process includes the following:

- 1003 1. The **Get_entropy_bitstring** process **shall** only be used to access one or more validated
1004 entropy sources. Non-validated entropy sources **shall** be accessed by a separate process
1005 to avoid possible misuse.
- 1006 2. Each validated entropy source **shall** be independent of all other validated or non-
1007 validated entropy sources used by the RBG.
- 1008 3. The output produced from multiple entropy-source calls to a single validated entropy
1009 source or by calls to multiple independent, validated entropy sources **shall** be
1010 concatenated into a single bitstring. The entropy in the bitstring is the sum of the entropy
1011 provided by the validated entropy sources that are to be credited for contributing entropy
1012 to the process. For Method 1 (see Sec. 2.3), only entropy contributed by one or more
1013 validated physical entropy sources is counted. For Method 2, the entropy from all
1014 validated entropy sources is counted.
- 1015 4. If a failure is reported during the **Get_entropy_bitstring** process by any physical or non-
1016 physical entropy source whose entropy is counted toward fulfilling an entropy request,
1017 the **Get_entropy_bitstring** process **shall** behave as follows (note that a bitstring
1018 containing entropy **should not** have been provided by that entropy source when a failure
1019 was reported; see Sec. 2.6, item 10):
 - 1020 a. Method 1 is used for counting the entropy from one or more physical entropy
1021 sources:
 - 1022 1) If a physical entropy source reports a failure, the error **shall** be reported
1023 to the consuming application as soon as possible. Any entropy collected

1024 during the execution of the **Get_entropy_bitstring** process in which the
1025 error is reported **shall not** be used. This failed entropy source **shall not**
1026 be accessed to obtain entropy until the condition that caused the failure
1027 has been corrected and operational tests have been successfully passed.

1028 If multiple physical entropy sources are used, the report **shall** identify
1029 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.

1041 b. Method 2 in Sec. 2.3 is used for counting the entropy from one or more non-
1042 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.

1056 5. The **Get_entropy_bitstring** process **shall not** provide output for RBG operations unless
1057 the bitstring contains sufficient entropy to fulfill the entropy request.

1058 3.2. External Conditioning

1059 Entropy bits produced by one or more entropy sources are required for seeding and reseeding
1060 the DRBG in the RBG constructions specified in this document. Whether or not entropy-source

1061 output was conditioned within a validated entropy source prior to output, the entropy provided
1062 by the validated entropy source(s) may need to be conditioned prior to subsequent use by the
1063 RBG. For example:

- 1064 • The entropy source within an RBG2 or RBG3 construction (see Sec. 5 or 6, respectively) is
1065 used to seed and reseed its DRBG. The entropy source may, for example, produce
1066 bitstrings that are too long for the specific DRBG implementation.
- 1067 • Seed material with full entropy is required when the CTR_DRBG is implemented without
1068 a derivation function and an entropy source is used for seeding and reseeding the DRBG.
1069 If the entropy sources does not provide full-entropy output, the output needs to be
1070 conditioned prior to subsequent use by the DRBG to obtain full-entropy input for the
1071 DRBG.
- 1072 • When the root RBGC construction in a DRBG chain uses a full-entropy source as its initial
1073 randomness source (see Sec. 7), the output from the entropy source(s) may need to be
1074 conditioned to provide a full-entropy bitstring for seeding and reseeding the root (i.e., the
1075 entropy source itself may not provide full-entropy output).
- 1076 • If both physical and non-physical entropy sources are used to provide seed material, the
1077 entropy within the concatenated bitstring produced by these sources may not be
1078 distributed uniformly throughout the bitstring.

1079 Since this conditioning is performed outside an entropy source, the output is said to be *externally*
1080 *conditioned*.

1081 The conditioning function operates on a bitstring that is produced by the **Get_entropy_bitstring**
1082 process to produce an *entropy_bitstring*. Reasons to perform conditioning might include:

- 1083 • Reducing the bias in the *entropy_bitstring*,
- 1084 • Distributing entropy uniformly across the *entropy_bitstring*,
- 1085 • Reducing the length of the *entropy_bitstring* and compressing the entropy into a smaller
1086 bitstring, and/or
- 1087 • Ensuring the availability of full-entropy bits.

1088 When external conditioning is performed, a vetted conditioning function listed in SP 800-90B
1089 **shall** be used. Additional vetted conditioning functions may be approved in the future.

1090 3.2.1. Conditioning Function Calls

1091 The conditioning functions operate on bitstrings obtained using the **Get_entropy_bitstring**
1092 process (see Section 3.1) to obtain an *entropy_bitstring* from one or more entropy sources.

1093 The following format is used in Section 3.2.2 for a conditioning-function call:

1094 *conditioned_output_block*=**Conditioning_function**(*input_parameters*),

1095 where the *input_parameters* for the selected conditioning function are discussed in Sections
1096 3.2.1.2 and 3.2.1.3, and *conditioned_output_block* is the output returned by the conditioning
1097 function.

1098 **3.2.1.1. Keys Used in External Conditioning Functions**

1099 The **HMAC**, **CMAC**, and **CBC-MAC** vetted conditioning functions require the input of a *Key*
1100 of a specific length (*keylen*), depending on the conditioning function and its primitive. Unlike
1101 other cryptographic applications, keys used in these external conditioning functions do not
1102 require secrecy to accomplish their purpose, so they may be hard-coded, fixed, or all zeros.

1103 For the **CMAC** and **CBC-MAC** conditioning functions, the length of the key **shall** be an
1104 **approved** key length for the block cipher used (e.g., *keylen* = 128, 192, or 256 bits for AES).

1105 For the **HMAC** conditioning function, the length of the key **shall** be equal to the length of the
1106 hash function's output (i.e., *output_len*).

1107 **Table 2. Key lengths for the hash-based conditioning functions**

Hash Function	Length of the output (<i>output_len</i>) and key (<i>keylen</i>)
SHA-256, SHA-512/256, SHA3-256	256
SHA-384, SHA3-384	384
SHA-512, SHA3-512	512

1108 Using random keys may provide some additional security in case the input is more predictable
1109 than expected. Thus, these keys **should** be chosen randomly (e.g., by obtaining bits directly from
1110 the entropy source and inserting them into the key or by providing entropy-source bits to a
1111 conditioning function with a fixed key to derive the new key). Any entropy used to randomize the
1112 key **shall not** be used for any other purpose.

1113 **3.2.1.2. Hash Function-based Conditioning Functions**

1114 Conditioning functions may be based on **approved** hash functions.

1115 One of the following calls **shall** be used for external conditioning when the conditioning function
1116 is based on a hash function:

1117 1. Using an **approved** hash function directly:

1118 *conditioned_output_block* = **Hash**(*entropy_bitstring*),

1119 where the hash function operates on the *entropy_bitstring* provided as input.

1120 2. Using HMAC with an **approved** hash function:

1121 *conditioned_output_block* = **HMAC**(*Key*, *entropy_bitstring*),

1122 where HMAC operates on the *entropy_bitstring* using a *Key* determined as specified in
1123 Sec. 3.2.1.1.

1124 In both cases, the length of the conditioned output is equal to the length of the output block of
1125 the selected hash function (i.e., *output_len*).

1126 3. Using **Hash_df**, as specified in SP 800-90A:

1127 $\text{conditioned_output_block} = \text{Hash_df}(\text{entropy_bitstring}, \text{output_len}),$

1128 where the derivation function operates on the *entropy_bitstring* provided as input to
1129 produce a bitstring of *output_len* bits.

1130 3.2.1.3. Block Cipher-Based Conditioning Functions

1131 Conditioning functions may be based on **approved** block ciphers.⁸ TDEA **shall not** be used as the
1132 block cipher.

1133 For block-cipher-based conditioning functions, one of the following calls **shall** be used for
1134 external conditioning:

1135 1. Using CMAC (as specified in SP 800-38B) with an **approved** block cipher:

1136 $\text{conditioned_output_block} = \text{CMAC}(\text{Key}, \text{entropy_bitstring}),$

1137 where CMAC operates on the *entropy_bitstring* using a *Key* determined as specified in
1138 Sec. 3.2.1.1.

1139 2. Using CBC-MAC (specified in SP 800-90B) with an **approved** block cipher:

1140 $\text{conditioned_output_block} = \text{CBC-MAC}(\text{Key}, \text{entropy_bitstring}),$

1141 where CBC-MAC operates on the *entropy_bitstring* using a *Key* determined as specified
1142 in Sec. 3.2.1.1.

1143 CBC-MAC **shall** only be used as an external conditioning function under the following
1144 conditions:

1145 1. The length of the input is an integer multiple of the block size of the block cipher
1146 (e.g., a multiple of 128 bits for AES). No padding is done by CBC-MAC itself.⁹

1147 2. If the CBC-MAC conditioning function is used for the external conditioning of an
1148 entropy source output for CTR_DRBG instantiation or reseeding:

- 1149 • A personalization string **shall not** be used during instantiation.
- 1150 • Additional input **shall not** be used during the reseeding of the
1151 CTR_DRBG but may be used during the generate process.

1152 CBC-MAC is not approved for any use other than in an RBG.

1153 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.

1154 *conditioned_output_block = Block_cipher_df(entropy_bitstring, block_length),*
1155 where **Block_cipher_df** operates on the *entropy_bitstring* using a key specified within
1156 the function, and the *block_length* is 128 bits for AES.
1157 In all three cases, the length of the conditioned output is equal to the length of the output block
1158 (i.e., 128 bits for AES).

1159 **3.2.2. Using a Vetted Conditioning Function**

1160 There are several cases in which the use of an external conditioning function is required to
1161 prepare the entropy-source output for use by a DRBG mechanism. Section 3.2.2.1 provides a
1162 procedure for obtaining entropy from one or more entropy sources and subsequently processing
1163 it using an external conditioning function when full-entropy output is not required from the
1164 conditioning function (e.g., the conditioning function is used to compress the entropy into a
1165 shorter bitstring or to distribute the entropy across the output). Section 3.2.2.2 provides a
1166 procedure for obtaining full entropy from the entropy source(s) when needed. When full entropy
1167 is not required, either procedure may be used.

1168 **3.2.2.1. External Conditioning When Full Entropy is Not Required**

1169 The **Get_conditioned_input** procedure specified below iteratively requests entropy from the
1170 **Get_entropy_bitstring** process (represented as a **Get_entropy_bitstring** procedure; see Sec.
1171 2.8.2 and 3.1) and distributes the entropy in the newly acquired *entropy_bitstring* across the
1172 conditioning function's output block. The output of the **Get_conditioned_input** procedure is the
1173 concatenation of the conditioning function output blocks. The entire output of the
1174 **Get_conditioned_input** procedure **shall** be provided as input to the DRBG mechanism (i.e., the
1175 output of the **Get_conditioned_input** function **shall not** be truncated).

1176 Let *output_len* be the length of the conditioning function's output block.

1177 **Get_conditioned_input:**

1178 **Input:**

- 1179 1. *n*: The amount of entropy to be obtained.
- 1180 2. *counting_method*: The counting method to be used (i.e., either Method 1 or Method
1181 2, as described in Sec. 2.3).
- 1182 3. *target_entropy_source*: An optional parameter that indicates the specific entropy
1183 source to be queried. If the *target_entropy_source* is not indicated, output is to be
1184 obtained from any validated entropy sources producing output that have not
1185 reported a failure.

1186 **Output:**

- 1187 1. *status*: The status returned from the **Get_conditioned_input** process.

1188 2. *Conditioned_entropy_bitstring*: A bitstring containing conditioned entropy or the *Null*
1189 string.

1190 **Process:**

- 1191 1. $v = \lceil n/\text{output_len} \rceil$.
- 1192 2. $w = \lceil n/v \rceil$.
- 1193 3. *Conditioned_entropy_bitstring* = the *Null* string.
- 1194 4. For $i = 1, \dots, v$
 - 1195 4.1 $(status, \text{entropy_bitstring}) = \text{Get_entropy_bitstring}(w, \text{counting_method},$
1196 *target_entropy_source*).
 - 1197 4.2 If $(status \neq \text{SUCCESS})$, then return $(status, \text{Null})$.
 - 1198 4.3 *conditioned_output_block* = **Conditioning_function**(*input_parameters*).
 - 1199 4.4 *Conditioned_entropy_bitstring* = *Conditioned_entropy_bitstring* ||
1200 *conditioned_output_block*.
- 1201 5. Return $(\text{SUCCESS}, \text{Conditioned_entropy_bitstring})$.

1202 Step 1 determines the number of output blocks (v) required to hold the requested amount of
1203 entropy.

1204 Step 2 determines the amount of entropy (w) that will be requested for each of the v output
1205 blocks.

1206 Step 3 sets the bitstring into which conditioned output will be collected (i.e.,
1207 *Conditioned_entropy_bitstring*) to the *Null* string.

1208 Step 4 is iterated v times to obtain and condition the requested amount of entropy for each
1209 output block of the conditioning function.

- 1210 • Step 4.1 requests w bits of entropy from the entropy source(s) using the
1211 **Get_entropy_bitstring** call (see Sec. 2.8.2 and 3.1), indicating the method to be used for
1212 counting entropy (i.e., Method 1 or Method 2) and (if provided as input) the entropy
1213 source to be used (indicated by the *target_entropy_source* input parameter).
- 1214 • Step 4.2 checks whether the *status* returned in step 4.1 indicated a success. If the *status*
1215 did not indicate a success, the *status* is returned with a *Null* string as the
1216 *Conditioned_entropy_bitstring*.
- 1217 • Step 4.3 invokes the conditioning function for processing the *entropy_bitstring* obtained
1218 from step 4.1 to distribute the entropy throughout the conditioning function's output
1219 block. The *input_parameters* for the selected **Conditioning_function** are specified in Sec.
1220 3.2.1.2 and 3.2.1.3 based on the conditioning function used.
- 1221 • Step 4.4 concatenates the *conditioned_output_block* from step 4.3 to the
1222 *Conditioned_entropy_bitstring*.

- 1223 • If all the requested entropy has not been obtained and conditioned, then go to step 4.1
1224 with an updated value of v .

1225 Step 5 returns a *status* of SUCCESS and the value of *Conditioned_entropy_bitstring*.

1226 **3.2.2.2. Conditioning Function to Obtain Full-Entropy Bitstrings**

1227 The **Get_conditioned_full_entropy_input** procedure specified below produces a bitstring with
1228 full entropy using one of the conditioning functions identified in Sec. 3.2.1 whenever a bitstring
1229 with full entropy is required. This process is unnecessary if full-entropy output is provided by the
1230 the entropy source(s).

1231 The approach used by this procedure is to acquire sufficient entropy from the entropy source(s)
1232 to iteratively produce *output_len* bits with full entropy in the conditioning function's output block,
1233 where *output_len* is the length of the output block. The amount of entropy required for each use
1234 of the conditioning function is *output_len* + 64 bits (see item 11 in Sec. 2.6). This process is
1235 repeated until the requested number of full-entropy bits has been produced.

1236 The **Get_conditioned_full_entropy_input** procedure obtains entropy from either 1) a
1237 designated entropy source (if a specific entropy source is identified as the *target_entropy_source*)
1238 or 2) any available entropy source using the **Get_entropy_bitstring** process (represented as a
1239 **Get_entropy_bitstring** procedure; see Sec. 2.8.2 and 3.1) and conditions the newly acquired
1240 *entropy_bitstring* to provide an *n*-bit string with full entropy.

1241 **Get_conditioned_full_entropy_input:**

1242 **Input:**

- 1243 1. *n*: The amount of entropy to be obtained.
- 1244 2. *counting_method*: The counting method to be used (i.e., either Method 1 or Method
1245 2, as described in Sec. 2.3).
- 1246 3. *target_entropy_source*: An optional parameter that indicates the specific entropy
1247 source to be queried. If the *target_entropy_source* is not indicated, output is to be
1248 obtained from any validated entropy sources producing output that have not
1249 reported a failure.

1250 **Output:**

- 1251 1. *status*: The status returned from the **Get_conditioned_full_entropy_input** process.
- 1252 2. *Full_entropy_bitstring*: An *n*-bit string with full entropy or the *Null* string.

1253 **Process:**

- 1254 1. *temp* = the *Null* string.
- 1255 2. *ctr* = 0.
- 1256 3. While *ctr* < *n*, do

1257 3.1 $(status, entropy_bitstring) = \text{Get_entropy_bitstring}(\underline{output_len} + 64,$
1258 *counting_method, target_entropy_source*).

1259 3.2 If ($status \neq \text{SUCCESS}$), then return ($status, Null$).

1260 3.3 $conditioned_output_block = \text{Conditioning_function}(input_parameters)$.

1261 3.4 $temp = temp \parallel conditioned_output_block$.

1262 3.5 $ctr = ctr + output_len$.

1263 4. $Full_entropy_bitstring = \text{leftmost}(temp, n)$.

1264 5. Return ($\text{SUCCESS}, Full_entropy_bitstring$).

1265 Steps 1 and 2 initialize the temporary bitstring ($temp$) for storing the full-entropy bitstring being
1266 assembled and the counter (ctr) that counts the number of full-entropy bits produced.

1267 Step 3 obtains and processes the entropy for each iteration.

- 1268 • Step 3.1 requests $output_len + 64$ bits of entropy from the validated entropy source(s)
1269 using the indicated method for counting entropy (i.e., Method 1 or Method 2) and (if
1270 present) using only the entropy source identified as the *target_entropy_source*. If the
1271 entropy source to be used is not identified, the entropy is to be obtained from all available
1272 entropy sources that have not reported a failure.
 - 1273 • Step 3.2 checks whether the *status* returned in step 3.1 indicated a success. If the *status*
1274 did not indicate a success, the *status* is returned along with a *Null* bitstring as the
1275 *Full_entropy_bitstring*.
 - 1276 • Step 3.3 invokes the conditioning function for processing the *entropy_bitstring* obtained
1277 from step 3.1. The *input_parameters* for the selected **Conditioning_function** are
1278 specified in Sec. 3.2.1.2 or 3.2.1.3, depending on the conditioning function used.
 - 1279 • Step 3.4 concatenates the *conditioned_output_block* received in step 3.3 to the temporary
1280 bitstring (*temp*).
 - 1281 • Step 3.5 increments the counter for the number of full-entropy bits that have been
1282 produced so far.
 - 1283 • If less than n full-entropy bits have been produced, repeat the process starting at step 3.1.
- 1284 Step 4 truncates the full-entropy bitstring to n bits.
- 1285 • Step 5 returns an n -bit full-entropy bitstring as the *Full_entropy_bitstring*.

1286 **4. RBG1 Construction Based on RBGs With Physical Entropy Sources**

1287 An RBG1 construction provides a source of cryptographic random bits from a device that has no
1288 internal randomness source. Its security depends entirely on its DRBG being instantiated securely
1289 from an RBG with access to a physical entropy source that resides outside of the device.

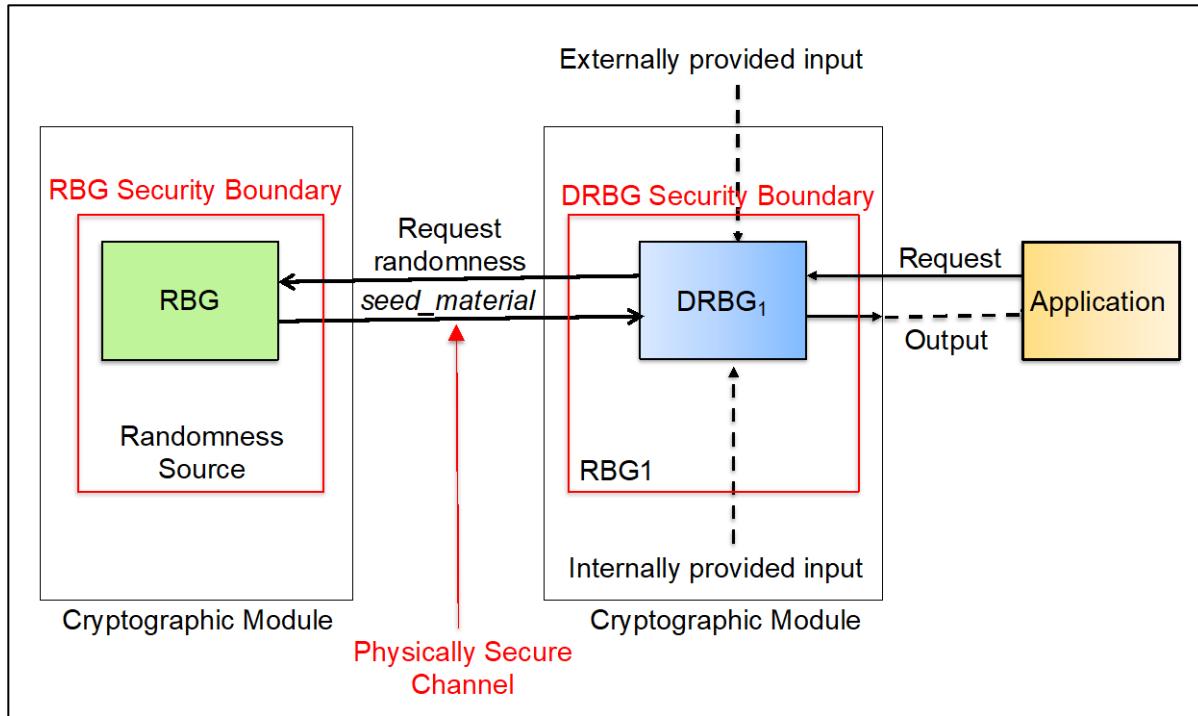
1290 The DRBG in an RBG1 construction is instantiated (i.e., seeded) only once using either an RBG2(P)
1291 construction (see Sec. 5) or an RBG3 construction (see Sec. 6). Since a randomness source is not
1292 available after DRBG instantiation, the DRBG within an RBG1 construction cannot be reseeded
1293 (i.e., prediction resistance and recovery from a compromise cannot be provided).

1294 An RBG1 construction may be useful for constrained devices in which an entropy source cannot
1295 be implemented or in any device in which access to a suitable source of randomness is not
1296 available after instantiation. Since the DRBG within an RBG1 construction cannot be reseeded,
1297 the use of the DRBG is limited to the DRBG's seedlife (see SP 800-90A).

1298 Optionally, subordinate DRBGs (i.e., sub-DRBGs) may be used within the security boundary of an
1299 RBG1 construction (see Sec. 4.3). The use of one or more sub-DRBGs may be useful for
1300 implementations that use flash memory, such as when the number of write operations to the
1301 memory is limited (resulting in short device lifetimes) or when there is a need to use different
1302 DRBG instantiations for different purposes. The DRBG in the RBG1 construction is the source of
1303 the randomness that is used to instantiate one or more sub-DRBGs. Each sub-DRBG is a DRBG
1304 specified in SP 800-90A and is intended to be used for a limited time and a limited purpose, so
1305 reseeding of the DRBG within a sub-DRBG is not provided. A sub-DRBG may, in fact, be a different
1306 instantiation of the DRBG design implemented within the RBG1 construction (see Sec. 2.4.1).

1307 **4.1. RBG1 Description**

1308 As shown in Fig. 15, an RBG1 construction consists of a DRBG contained within a DRBG security
1309 boundary in one cryptographic module and an RBG (serving as a randomness source) contained
1310 within a separate cryptographic module from that of the RBG1 construction. For convenience
1311 and clarity, the DRBG within the RBG1 construction will sometimes be referred to as DRBG₁. Note
1312 that the required health tests are not shown in the figure.



1313

1314

Fig. 15. Generic structure of the RBG1 construction

1315 The RBG for instantiating DRBG₁ **must** be either an RBG2(P) construction that supports a reseed
1316 request from the RBG1 construction (see Sec. 5) or an RBG3 construction (see Sec. 6). A physically
1317 secure channel between the randomness source and DRBG₁ is used to securely transport the
1318 seed material required for DRBG instantiation. An optional recommended personalization string
1319 and optional additional input may be provided from within the DRBG's cryptographic module or
1320 from outside of that module (see Sec. 2.4.1).

1321 An external conditioning function is not needed for this design because the output of the RBG
1322 used as the randomness source has already been cryptographically processed. The output from
1323 an RBG1 construction may be used within the cryptographic module (e.g., to seed a sub-DRBG,
1324 as specified in Sec. 4.3) or by an application outside of the RBG1 security boundary. The security
1325 strength of the output produced by the RBG1 construction is the minimum of the security
1326 strengths provided by the DRBG within the construction and the RBG used as the randomness
1327 source to seed the DRBG. Examples of RBG1 and sub-DRBG constructions are provided in
1328 Appendices B.2 and B.3, respectively.

1329 **4.2. Conceptual Interfaces**

1330 Interfaces to the DRBG within an RBG1 construction include requests for instantiating the DRBG
1331 and generating pseudorandom bits (see Sec. 4.2.1 and 4.2.2, respectively). A reseed of the RBG1
1332 construction cannot be performed because the randomness source is not available after
1333 instantiation.

1334 **4.2.1. Instantiating the DRBG in the RBG1 Construction**

1335 The DRBG within the RBG1 construction (DRBG_1) may be instantiated by an application at any
1336 security strength possible for the DRBG design using the **DRBG_Instantiate_request** discussed
1337 in Sec. 2.8.1.1:

1338 $(status, \text{RBG}_1_DRBG1_state_handle) =$
1339 **DRBG_Instantiate_request** ($s, personalization_string$).

1340 The **DRBG_Instantiate_request** received by DRBG_1 from an application **shall** result in the
1341 execution of the **DRBG_Instantiate** function within DRBG_1 (see Sec. 2.8.1.1):

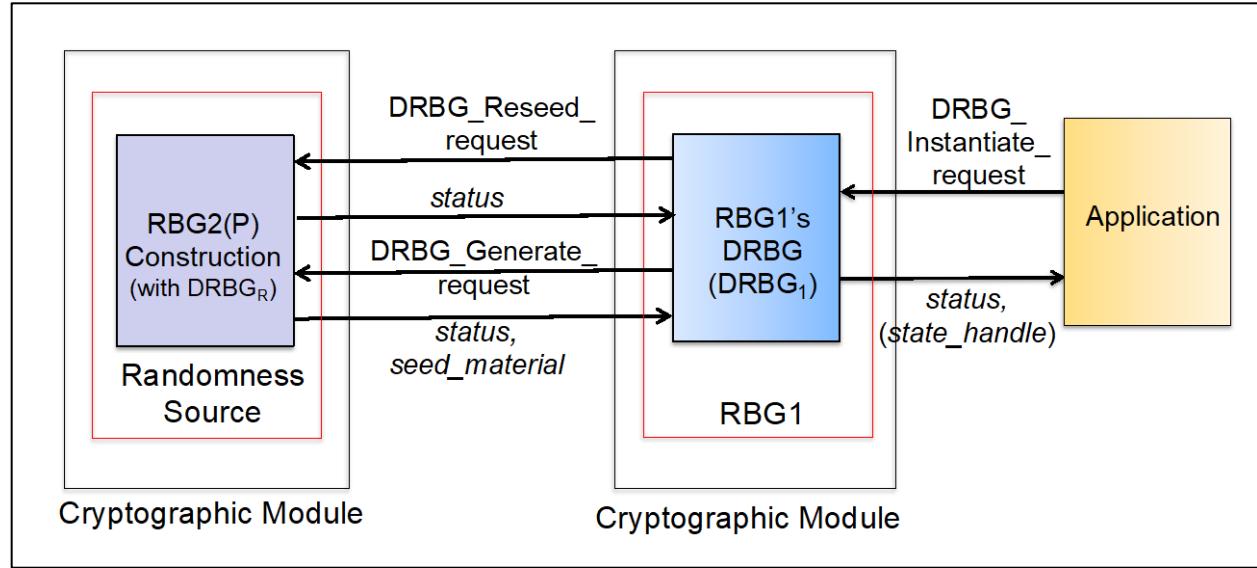
1342 $(status, \text{RBG}_1_DRBG1_state_handle) =$
1343 **DRBG_Instantiate** ($s, personalization_string$).

1344 The $status$ returned by the **DRBG_Instantiate** function **shall** be returned to the requesting
1345 application in response to the **DRBG_Instantiate_request**. $\text{RBG}_1_DRBG1_state_handle$ is the
1346 state handle for DRBG_1 's internal state; the state handle may be *Null*.

1347 The **DRBG_Instantiate** function within DRBG_1 **shall** use an external RBG (i.e., the randomness
1348 source) to obtain the *seed_material* necessary for establishing the DRBG's security strength.

1349 In SP 800-90A, the **DRBG_Instantiate** function specifies the use of a **Get_randomness-**
1350 **source_input** call to obtain seed material from the randomness source for instantiation (see Sec.
1351 2.8.1.4 in this document and SP 800-90A). For an RBG1 construction, an **approved** external
1352 RBG2(P) or RBG3 construction **must** be used as the randomness source (see Sec. 5 and 6,
1353 respectively).

1354 If the randomness source is an RBG2(P) construction (see Fig. 16), the RBG2(P) construction **must**
1355 be reseeded using its internal entropy source(s) before generating bits to be provided to DRBG_1 .
1356 The **Get_randomness-source_input** call in the **DRBG_Instantiate** function of DRBG_1 **shall** be
1357 replaced by a reseed request followed by a generate request to the RBG2(P) construction serving
1358 as the randomness source (see steps 1a and 2a below).

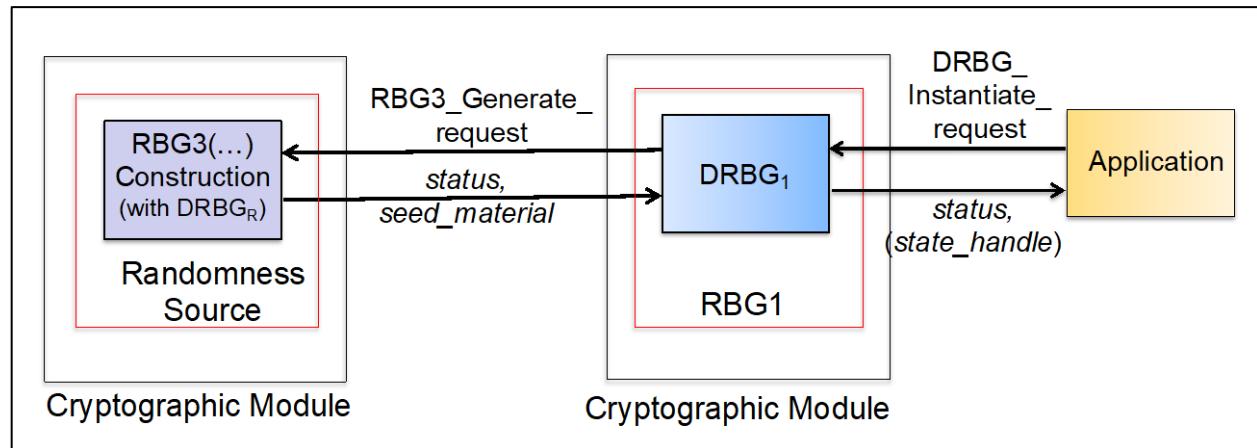


1359

1360

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

1361 If the randomness source is an RBG3 construction (as shown in Fig. 17), the **Get_randomness-**
1362 **source_input** call in the **DRBG_Instantiate** function of DRBG₁ **shall** be replaced by the
1363 appropriate call to the RBG3 generate function (see Sec. 2.8.3.2, 6.4.1.2, and 6.5.1.2 and steps
1364 1b and 2b below).



1365

1366

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

1367 Let DRBG₁ be the DRBG to be instantiated within the RBG1 construction and let DRBG_R be the
1368 DRBG used within the randomness source (i.e., an RBG2(P) or RBG3 construction). Let *s* be the
1369 security strength to be instantiated for DRBG₁. **DRBG_Reseed_request** and
1370 **DRBG_Generate_request** are used below by an application to request the generation and
1371 reseed of the DRBG within the randomness source (i.e., DRBG_R). Let *DRBG_R_state_handle* be the
1372 state handle for DRBG_R.

1373 Upon receiving the instantiation request from the application, DRBG₁ is instantiated as follows:

1374 1. When an RBG1 construction is instantiating a CTR_DRBG without a derivation function,
1375 $s + 128$ bits¹⁰ **shall** be obtained from the randomness source as follows:

1376 a. If the randomness source is an RBG2(P) construction (see Fig. 16), the
1377 **Get_randomness-source_input** call in the **DRBG_Instantiate** function of DRBG₁
1378 is replaced by a request to reseed DRBG_R (the DRBG within the RBG2(P)
1379 construction), followed by a request to generate bits:

- 1380 • $status = \text{DRBG_Reseed_request}(DRBG_R_state_handle, additional_input)$.
- 1381 • If ($status \neq \text{SUCCESS}$), then return ($status, Invalid_state_handle$).
- 1382 • $(status, seed_material) = \text{DRBG_Generate_request}(DRBG_R_state_handle, s + 128, s, additional_input)$.
- 1383 • If ($status \neq \text{SUCCESS}$), then return ($status, Invalid_state_handle$).

1384

1385 **DRBG_Reseed_request** and **DRBG_Generate_request** are used here to
1386 indicate requests for the DRBG within the randomness source (DRBG_R) to execute
1387 the **DRBG_Reseed** function and **DRBG_Generate** function within DRBG_R (see
1388 Sec. 2.8.1.3, and 2.8.1.2, respectively). Also, see Sec. 5.2.3 and 5.2.2 for the
1389 handling of the reseed and generate requests by the RBG2(P) construction.

1390 b. If the randomness source is an RBG3(XOR) or RBG3(RS) construction (see Fig. 17),
1391 the **Get_randomness-source_input** call in the **DRBG_Instantiate** function of
1392 DRBG₁ is replaced by a request for the generation of random bits:

- 1393 • $(status, seed_material) = \text{RBG3_Generate_request}(DRBG_R_state_handle, s + 128, additional_input)$.
- 1394 • If ($status \neq \text{SUCCESS}$), then return ($status, Invalid_state_handle$).

1395

1396 **RBG3_Generate_request** is intended to result in the execution of the
1397 **DRBG_Generate** function in DRBG_R (see Sec. 2.8.3.1). Also, see Sec. 6.4.1.2 and
1398 6.5.1.2.1 for the handling of the generate request by the RBG3(XOR) and RBG3(RS)
1399 constructions, respectively.

1400 2. When an RBG1 construction is instantiating any other DRBG (including a CTR_DRBG
1401 with a derivation function¹¹), $3s/2$ bits **shall** be obtained from a randomness source that
1402 provides a security strength of at least s bits.

1403 a. If the randomness source is an RBG2(P) construction (see Fig. 16), the
1404 **Get_randomness-source_input** call in DRBG₁ is replaced by a request to reseed
1405 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.

- 1406 • $status = \text{DRBG_Reseed_request}(DRBG_R_state_handle, additional_input)$.
- 1407 • If ($status \neq \text{SUCCESS}$), then return ($status, Invalid_state_handle$).
- 1408 • $(status, seed_material) = \text{DRBG_Generate_request}(DRBG_R_state_handle, 3s/2, s, additional_input)$.
- 1409 • If ($status \neq \text{SUCCESS}$), then return ($status, Invalid_state_handle$).

1411 **DRBG_Reseed_request** and **DRBG_Generate_request** are used here to
1412 indicate requests for the DRBG within the randomness source (DRBG_R) to execute
1413 the **DRBG_Reseed** function and **DRBG_Generate** function within DRBG_R (see
1414 Sec. 2.8.1.3 and 2.8.1.2, respectively). Also, see Sec. 5.2.3 and 5.2.2 for the
1415 handling of the reseed and generate requests by the RBG2(P) construction.

- 1416 b. If the randomness source is an RBG3(XOR) or RBG3(RS) construction (see Fig. 17),
1417 the **Get_randomness-source_input** call in DRBG₁ is replaced by a request for the
1418 generation of random bits:

- 1419 • $(status, seed_material) =$
1420 **RBG3_DRBG_Generate_request**($DRBG_R_state_handle, 3s/2,$
1421 *additional_input*).
- 1422 • If ($status \neq \text{SUCCESS}$), then return ($status, Invalid_state_handle$).

1423 **RBG3_DRBG_Generate_request** is intended to result in the execution of the
1424 **DRBG_Generate function** in DRBG_R (see Sec. 2.8.3.1). Also, see Sec. 6.4.1.2 and
1425 6.5.1.2.1 for the handling of the generate request by the RBG3(XOR) and RBG3(RS)
1426 constructions, respectively.

1427 **4.2.2. Requesting Pseudorandom Bits**

1428 As discussed in Sec. 2.8.1.2, an application requests the RBG1 construction to generate bits as
1429 follows:

1430 $(status, returned_bits) = \text{DRBG_Generate_request}(RBG1_DRBG1_state_handle,$
1431 *requested_number_of_bits*, $s, additional_input$).

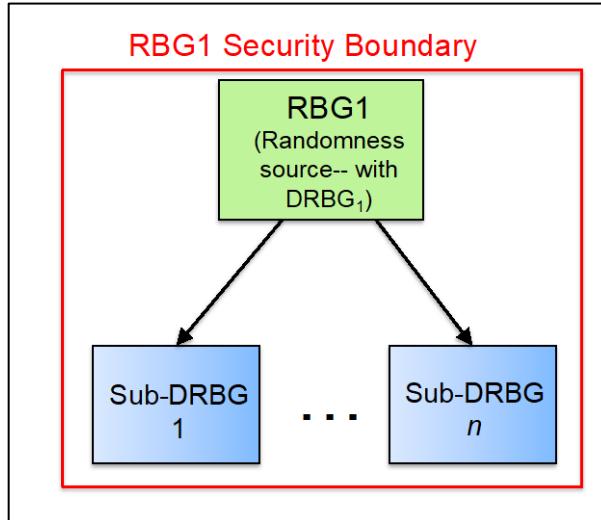
1432 The **DRBG_Generate_request** results in the execution of the **DRBG_Generate** function within
1433 DRBG₁:

1434 $(status, returned_bits) = \text{DRBG_Generate}(RBG1_DRBG1_state_handle,$
1435 *requested_number_of_bits*, $s, additional_input$).

1436 The *status* returned by the **DRBG_Generate** function **shall** be returned to the requesting
1437 application. If the *status* indicates a successful process, the *returned_bits* **shall** also be provided
1438 to the application in response to the request.

1439 **4.3. Using an RBG1 Construction With Subordinate DRBGs (Sub-DRBGs)**

1440 Figure 18 depicts an example of the use of optional subordinate DRBGs (sub-DRBGs) within the
1441 security boundary of an RBG1 construction. The RBG1 construction is used as the randomness
1442 source to provide separate outputs to instantiate each of its sub-DRBGs.



1443

1444 **Fig. 18. RBG1 construction with sub-DRBGs**

1445 The RBG1 construction and each of its sub-DRBGs **shall** be implemented as separate physical or
1446 logical entities (see Fig. 18). Let DRBG₁ be the DRBG used by the RBG1 construction itself, with
1447 *RBG1_DRBG1_state_handle* used as the state handle for the internal state of DRBG₁. Let *sub-*
1448 *DRBGx_state_handle* be the state handle for the internal state of sub-DRBGx.

- 1449 • When implemented as separate physical entities, the DRBG algorithms used by DRBG₁
1450 and the sub-DRBGs **shall** be the same DRBG algorithm (e.g., the RBG1 construction and
1451 all its sub-DRBGs use HMAC_DRBG with SHA-256).
- 1452 • When implemented as separate logical entities, the same software or hardware
1453 implementation of a DRBG algorithm is used but with a different internal state for each
1454 logical entity.

1455 The sub-DRBGs have the following characteristics:

- 1456 1. Only one layer of sub-DRBGs is allowed.
- 1457 2. Sub-DRBG outputs are considered outputs of the RBG1 construction.
- 1458 3. The security strength that can be provided by a sub-DRBG is no more than the security
1459 strength of DRBG₁ (i.e., the DRBG within the RBG1 construction that is serving as the
1460 randomness source for the sub-DRBG).
- 1461 4. Sub-DRBGs cannot provide output with full entropy.
- 1462 5. The number of sub-DRBGs that can be instantiated by an RBG1 construction is limited
1463 only by the practical considerations associated with the implementation or application.

1464 **4.3.1. Instantiating a Sub-DRBG**

1465 An application may request the DRBG₁ construction to instantiate a sub-DRBG. The following
1466 represents the form of the application's request for sub-DRBG instantiation:

1467 $(status, sub\text{-}DRBG_state_handle) =$
1468 **Instantiate_sub-DRBG_request**(*s, personalization_string*).

1469 DRBG₁ executes an **Instantiate_sub-DRBG** function. The *status* of the process is returned to the
1470 application with a state handle if the *status* indicates success.

1471 The value of *max_personalization_string_length* is specified in SP 800-90A.

1472 **Instantiate_sub-DRBG:**

1473 **Input:**

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

1476 **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).

1482 **Information retained within the DRBG boundary after instantiation:**

1483 The internal states for DRBG₁ and the sub-DRBG instantiation.

1484 **Process:**

1. Obtain the current internal state of DRBG₁ 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* ≤ 128), then *s* = 128.
Else *s* = 192.

1494 Comment: See the instructions below for the value
1495 of *number_of_bits_to_generate*.

- 1496 5. $(status, seed_material) = \mathbf{DRBG_Generate}(RBG1_DRBG1_state_handle,$
1497 $number_of_bits_to_generate, s)$.
 - 1498 6. If $(status \neq \text{SUCCESS})$, return $(status, Invalid_state_handle)$.
 - 1499 7. $working_state_values = \mathbf{Instantiate_algorithm}(seed_material,$
1500 $personalization_string)$.
 - 1501 8. Get the $sub\text{-}DRBG_state_handle$ for a currently empty internal state. If an empty
1502 internal state cannot be found, return $(\text{ERROR_FLAG}, Invalid_state_handle)$.
 - 1503 9. Set the internal state for the new instantiation (e.g., as indicated by
1504 $sub\text{-}DRBG_state_handle$):
 - 1505 9.1 Record the $working_state_values$ returned from step 7.
 - 1506 9.2 Record any administrative information (e.g., the value of s).
 - 1507 10. Return $(\text{SUCCESS}, sub\text{-}DRBG_state_handle)$.
- 1508 Step 1 obtains DRBG₁'s security strength. A description of the internal state for each DRBG type
1509 is provided in SP 800-90A.
- 1510 Steps 2 and 3 check the validity of the requested security strength s and the length of any
1511 personalization string provided for the instantiation request. An ERROR_FLAG and an invalid
1512 state handle are returned to the requesting application if either is unacceptable.
- 1513 Step 4 sets the security strength to be established for the sub-DRBG instantiation based on the
1514 requested security strength s .
- 1515 Step 5 requests the generation of $seed_material$ at a security strength of s bits using DRBG₁. The
1516 $number_of_bits_to_generate$ depends on DRBG₁'s type:
 - 1517 • When CTR_DRBG without a derivation function is implemented for DRBG₁,
1518 $number_of_bits_to_generate = s + 128$.
 - 1519 • Otherwise, $number_of_bits_to_generate = 3s/2$.
- 1520 Step 6 checks the $status$ returned from step 5. If a $status$ of SUCCESS is not returned, the $status$
1521 and an invalid state handle are returned to the requesting application.
- 1522 Step 7 invokes the appropriate instantiate algorithm in SP 800-90A for DRBG₁'s design. Values for
1523 the working state portion of the sub-DRBG's internal state are returned by the instantiate
1524 algorithm.
- 1525 Step 8 assigns a state handle for an available internal state. If no internal state is currently
1526 available, an ERROR_FLAG and invalid state handle are returned to the requesting application.
- 1527 Step 9 enters the required values into the assigned internal state for the sub-DRBG.
- 1528 Step 10 returns a $status$ of SUCCESS and the assigned state handle to the requesting application.

1529 **4.3.2. Requesting Random Bits From a Sub-DRBG**

1530 As discussed in Sec. 2.8.1.2, pseudorandom bits may be requested from a sub-DRBG by an
1531 application:

1532 $(status, returned_bits) = \text{DRBG_Generate request}(sub_DRBGx_state_handle,$
1533 $requested_number_of_bits, requested_security_strength, additional_input)$.

1534 The generate request received by the sub-DRBG **shall** result in the execution of the
1535 **DRBG_Generate** function:

1536 $(status, returned_bits) = \text{DRBG_Generate}(sub_DRBGx_state_handle,$
1537 $requested_number_of_bits, requested_security_strength, additional_input)$.

1538 The *status* returned by the **DRBG_Generate** function **shall** be returned to the application in
1539 response to the request. If the process is successful, the newly generated bits (*returned_bits*)
1540 **shall** also be provided to the application in response to the **DRBG_Generate_request**.

1541 **4.4. Requirements**

1542 **4.4.1. RBG1 Construction Requirements**

1543 An RBG1 construction being instantiated has the following testable requirements (i.e., testable
1544 by the validation labs):

- 1545 1. An **approved** DRBG from SP 800-90A whose components can provide the targeted
1546 security strength for the RBG1 construction **shall** be employed.
- 1547 2. The components of the RBG1 construction **shall** be successfully validated for compliance
1548 with SP 800-90A, SP 800-90C, FIPS 140, and the specification of any other **approved**
1549 algorithm used within the RBG1 construction, as applicable.
- 1550 3. The RBG1 construction **shall not** produce any output until it is instantiated.
- 1551 4. The RBG1 construction **shall not** include a capability to be reseeded.
- 1552 5. The RBG1 construction **shall not** permit itself to be instantiated more than once.¹²
- 1553 6. The randomness source **shall** be in a separate device from that of the RBG1 construction.
- 1554 7. For CTR_DRBG (with a derivation function), Hash_DRBG, or HMAC_DRBG, $3s/2$ bits
1555 **shall** be obtained from a randomness source, where s is the targeted security strength for
1556 the DRBG used in the RBG1 construction (DRBG₁).

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.

- 1557 8. For a CTR_DRBG without a derivation function within the RBG1 construction, $s + 128$
1558 bits¹³ **shall** be obtained from the randomness source, where s is the targeted security
1559 strength for the DRBG used in the RBG1 construction (DRBG₁).
1560 9. An implementation of an RBG1 construction **shall** verify that the internal state has been
1561 updated before the generated output is provided to the requesting entity.
1562 10. The RBG1 construction **shall not** provide output for generating requests that specify a
1563 security strength greater than the instantiated security strength of its DRBG.
1564 11. If the RBG1 construction can be used to instantiate a sub-DRBG, the RBG1 construction
1565 **may** directly produce output for an application in addition to instantiating a sub-DRBG.
1566 12. Seed material produced by the RBG1 construction to instantiate a sub-DRBG **shall not** be
1567 used to instantiate other sub-DRBGs nor be provided directly to a consuming application.
1568 13. If the seedlife of the DRBG within the RBG1 construction (DRBG₁) is ever exceeded or a
1569 health test of the DRBG fails, the use of the RBG1 construction **shall** be terminated.
1570 The non-testable requirements for the RBG1 construction are listed below. If these requirements
1571 are not met, no assurance can be obtained about the security of the implementation.
1572 14. A validated RBG2(P) construction with support for reseeding requests or a validated RBG3
1573 construction **must** be used as the randomness source for the DRBG in the RBG1
1574 construction (DRBG₁).
1575 15. The randomness source **must** provide the requested number of bits at a security strength
1576 of s bits or higher, where s is the targeted security strength for the DRBG within the RBG1
1577 construction (DRBG₁).
1578 16. The specific output of the randomness source (or portion thereof) that is used for the
1579 instantiation of an RBG1 construction **must not** be used for any other purpose, including
1580 for seeding a different instantiation.
1581 17. If an RBG2(P) construction is used as the randomness source for the RBG1 construction,
1582 the RBG2(P) construction **must** be reseeded before generating bits for each RBG1
1583 instantiation.
1584 18. A physically secure channel **must** be used to insert the seed material from the
1585 randomness source into the DRBG of the RBG1 construction (DRBG₁).

1586 **4.4.2. Sub-DRBG Requirements**

- 1587 A sub-DRBG has the following testable requirements (i.e., testable by the validation labs):
1588 1. The randomness source for a sub-DRBG **shall** be an RBG1 construction, and a sub-DRBG
1589 **shall not** serve as a randomness source for another sub-DRBG.

¹³ Note that $s + 128 = \text{keylen} + \text{blocklen} = \text{seedlen}$, as specified in SP 800-90Ar1.

- 1590 2. A sub-DRBG **shall** employ the same DRBG components as its randomness source (i.e., the
1591 RBG1 construction).
- 1592 3. A sub-DRBG **shall** reside in the same security boundary as the RBG1 construction that
1593 instantiates it.
- 1594 4. The output from the RBG1 construction that is used for sub-DRBG instantiation **shall not**
1595 be output from the security boundary that contains the RBG1 construction and sub-DRBG
1596 and **shall not** be used for any other purpose, including for seeding a different sub-DRBG.
- 1597 5. The security strength for a target sub-DRBG **shall not** exceed the security strength that is
1598 supported by the RBG1 construction.
- 1599 6. For CTR_DRBG (with a derivation function), Hash_DRBG, or HMAC_DRBG, $3s/2$ bits
1600 **shall** be obtained from the RBG1 construction for instantiation of the sub-DRBG, where s
1601 is the requested security strength for the target sub-DRBG.
- 1602 7. For a CTR_DRBG without a derivation function used by the sub-DRBG, $s + 128$ bits **shall**
1603 be obtained from the RBG1 construction for instantiation, where s is the requested
1604 security strength for the target sub-DRBG.
- 1605 8. A sub-DRBG **shall not** produce output until it is instantiated.
- 1606 9. A sub-DRBG **shall not** provide output for generating requests that specify a security
1607 strength greater than the instantiated security strength of the sub-DRBG.
- 1608 10. An implementation of a sub-DRBG **shall** verify that the internal state has been updated
1609 before the generated output is provided to the requesting entity.
- 1610 11. The sub-DRBG **shall not** be reseeded.
- 1611 12. If the seedlife of a sub-DRBG is ever exceeded or a health test of the sub-DRBG fails, the
1612 use of the sub-DRBG **shall** be terminated.
- 1613 A non-testable requirement for a sub-DRBG (i.e., not testable by the validation labs) is:
- 1614 13. The output of a sub-DRBG **must not** be used as seed material for other DRBGs (e.g., the
1615 DRBGs in other RBGs) or sub-DRBGs.

1616 **5. RBG2 Constructions Based on Physical and/or Non-Physical Entropy Sources**

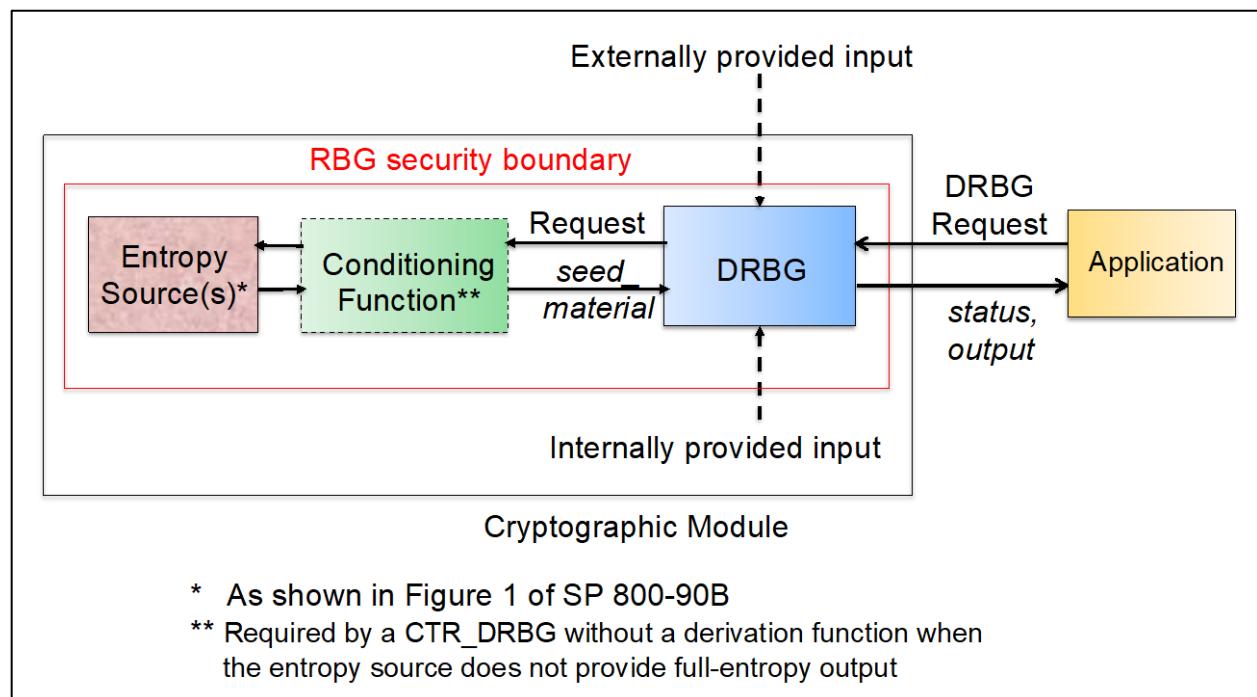
1617 An RBG2 construction is a cryptographically secure RBG with continuous access to one or more
 1618 validated entropy sources within its RBG security boundary. The RBG is instantiated before use
 1619 and generates outputs on demand. An RBG2 construction may (optionally) be implemented to
 1620 support reseeding requests from a consuming application (i.e., providing prediction resistance
 1621 for the next output of the RBG2 construction to mitigate a possible compromise of previous
 1622 internal states) and/or to be reseeded in accordance with implementation-selected criteria.

1623 If a consuming application requires full-entropy output, an RBG3 construction from Sec. 6 needs
 1624 to be used rather than an RBG2 construction.

1625 An RBG2 construction may be useful for all devices in which an entropy source can be
 1626 implemented.

1627 **5.1. RBG2 Description**

1628 The DRBG for an RBG2 construction is contained within the same RBG security boundary and
 1629 cryptographic module as its validated entropy source(s) (see Fig. 19). One or more entropy
 1630 sources are used to provide the entropy bits for both DRBG instantiation and any reseeding of
 1631 the DRBG. The use of a personalization string and additional input is optional and may be
 1632 provided from within the cryptographic module or from outside of that module.



1633

1634 **Fig. 19. Generic structure of the RBG2 construction**

1635 The output from the RBG may be used within the cryptographic module or by an application
 1636 outside of the module.

1637 An example of an RBG2 construction is provided in Appendix B.4.

1638 An RBG2 construction may be implemented to use one or more validated physical and/or non-
1639 physical entropy sources for instantiation and reseeding. Two variants of the RBG2 construction
1640 may be implemented:

- 1641 1. An RBG2(P) construction uses the output of one or more validated physical entropy
1642 sources and (optionally) one or more validated non-physical entropy sources, as discussed
1643 in Method 1 of Sec. 2.3 (i.e., only the entropy produced by one or more validated physical
1644 entropy sources is counted toward the entropy required for instantiating or reseeding the
1645 RBG). Any amount of entropy may be obtained from a non-physical entropy source as
1646 long as sufficient entropy has been obtained from the physical entropy sources to fulfill
1647 an entropy request. An RBG2(P) construction may exist as part of an RBG3 construction
1648 (see Sec. 6).
- 1649 2. An RBG2(NP) construction uses the output of any validated non-physical or physical
1650 entropy source(s), as discussed in Method 2 of Sec. 2.3 (i.e., the entropy produced by both
1651 validated physical and non-physical entropy sources is counted toward the entropy
1652 required for instantiating or reseeding the RBG).

1653 These variants may affect the implementation of a **Get_entropy_bitstring** process (represented
1654 as a **Get_entropy_bitstring** procedure; see Sec. 2.8.2 and 3.1), either accessing the entropy
1655 source(s) directly or via the **Get_conditioned_input** or **Get_conditioned_full_entropy_input**
1656 procedure specified in Sec. 3.2.2 during instantiation and reseeding (see Sec. 5.2.1 and 5.2.3).
1657 That is, when seeding and reseeding an RBG2(P) construction (including a DRBG within an RBG3
1658 construction, as discussed in Sec. 6), Method 1 in Sec. 2.3 is used to combine the entropy from
1659 the entropy source(s), and Method 2 is used when instantiating and reseeding an RBG2(NP)
1660 construction.

1661 **5.2. Conceptual Interfaces**

1662 The RBG2 construction includes requests for instantiating the DRBG (see Sec. 5.2.1) and
1663 generating pseudorandom bits (see Sec. 5.2.2). Once instantiated, an RBG2 construction may be
1664 reseeded when requested by a consuming application or when determined by implementation-
1665 selected criteria if a reseed capability has been implemented (see Sec. 5.2.3).

1666 **5.2.1. RBG2 Instantiation**

1667 An RBG2 construction may be instantiated by an application at any valid¹⁴ security strength
1668 possible for the DRBG design and its components using an instantiation request (see Sec. 2.8.1.1):
1669 $(status, RBG2_DRBG_state_handle) = \text{DRBG_Instantiate_request}(s, personalization_string)$.

1670 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.

1671 $(status, RBG2_DRBG_state_handle) = \text{DRBG_Instantiate}(s, personalization_string)$.

1672 The **DRBG_Instantiation** function returns the *status* of the process, which is then provided to
1673 the application in response to the request. If the process is successful, a state handle for the
1674 instantiation (e.g., *RBG2_DRBG_state_handle*) is also returned from the **DRBG_Instantiate**
1675 function and may be forwarded to the application.¹⁵

1676 An RBG2 construction obtains entropy for its DRBG from one or more validated entropy sources
1677 within its boundary, either directly or using a conditioning function to obtain and process the
1678 output of the entropy source(s).

1679 SP 800-90A uses a **Get_randomness-source_input** call in the **DRBG_Instantiate** function to
1680 obtain the entropy needed for instantiation. Let *counting_method* indicate the method for
1681 counting entropy from the entropy source(s) (i.e., Method 1 counts only entropy provided by
1682 physical entropy sources, and Method 2 counts entropy from non-physical and physical entropy
1683 sources; see Sec. 2.3).

1684 1. When the DRBG is a CTR_DRBG without a derivation function, full-entropy bits **shall** be
1685 obtained from the entropy source(s) as follows:

1686 a. If all entropy sources provide full-entropy output, the **Get_randomness-**
1687 **source_input** call is replaced by:

- $(status, seed_material) = \text{Get_entropy_bitstring}(s + 128,$
1689 *counting_method*).¹⁶
- If $(status \neq \text{SUCCESS})$, then return $(status, Invalid_state_handle)$.

1690 The output of the entropy source(s) **shall** be concatenated to obtain the $s + 128$
1691 full-entropy bits to be returned as *seed_material*.

1693 b. If one or more entropy sources do not provide full-entropy output, the
1694 **Get_randomness-source_input** call is replaced by:¹⁷

- $(status, seed_material) = \text{Get_conditioned_full_entropy_input}(s + 128,$
1696 *counting_method*).
- If $(status \neq \text{SUCCESS})$, then return $(status, Invalid_state_handle)$.

1698 3. For CTR_DRBG (with a derivation function), Hash_DRBG, or HMAC_DRBG used as
1699 the DRBG, the entropy source(s) **shall** provide $3s/2$ bits of entropy to establish the security
1700 strength.

¹⁵ If there is never more than one DRBG instantiation possible, then a state handle is not required.

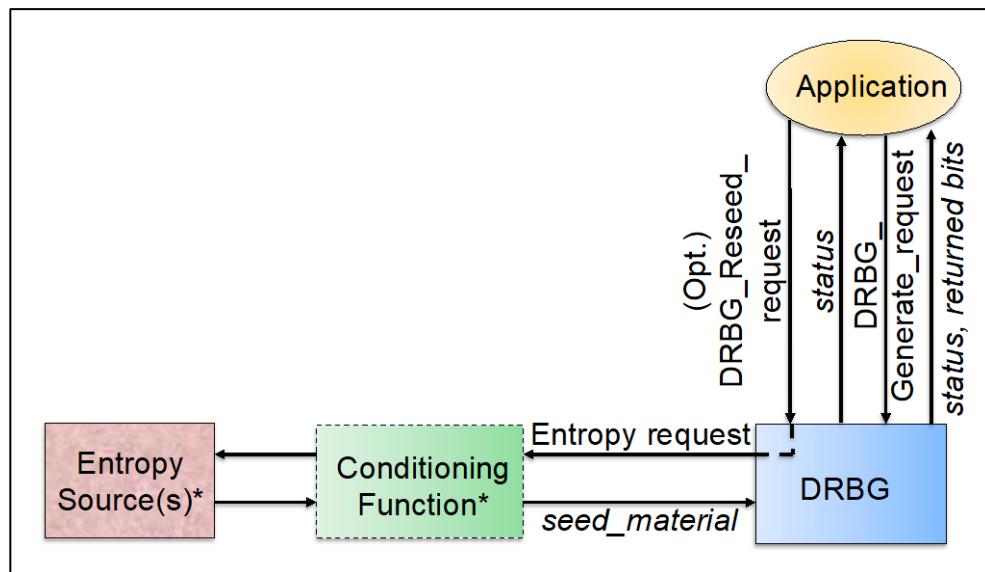
¹⁶ For a CTR_DRBG using AES, $s + 128 =$ the length of the key + the length of the AES block = *seedlen* (see Table 2 in SP 800-90Ar1).

¹⁷ See Sec. 3.2.2.2 for a specification of the **Get_conditioned_full_entropy_input** function.

- 1701 a. If the implementer wants full entropy in the bitstring to be provided to the DRBG,
 1702 the **Get_randomness-source_input** call is replaced by:
- 1703 • $(status, seed_material) = \text{Get_conditioned_full_entropy_input}(3s/2,$
 1704 $counting_method)$.
- 1705 • If $(status \neq \text{SUCCESS})$, then return $(status, Invalid_state_handle)$.
- 1706 b. Otherwise, the **Get_randomness-source_input** call is replaced by either:
- 1707 • $(status, seed_material) = \text{Get_entropy_bitstring}(3s/2, counting_method)$
- 1708 OR
- 1709 $(status, seed_material) = \text{Get_conditioned_input}(3s/2, counting_method)$.
- 1710 • If $(status \neq \text{SUCCESS})$, then return $(status, Invalid_state_handle)$.

1711 5.2.2. Requesting Pseudorandom Bits From an RBG2 Construction

1712 If prediction resistance is desired by a consuming application for the next RBG output to be
 1713 generated so that previous internal states that may have been compromised cannot be used to
 1714 determine the next RBG output, the application requests a reseed of the DRBG as discussed in
 1715 Sec. 5.2.3 before requesting the generation of pseudorandom bits. Figure 20 depicts an (optional)
 1716 reseed request before requesting the generation of pseudorandom bits.



1717 1718 Fig. 20. RBG2 generate request following an optional reseed request

1719 If a reseed of the RBG was not requested by the application prior to requesting the generation of
1720 pseudorandom bits or a *status* of SUCCESS was returned by the **DRBG_Reseed** function in
1721 response to a reseed request, pseudorandom bits are requested as follows (see Sec. 2.8.1.2):

1722 $(status, returned_bits) = \text{DRBG_Generate_request}(RBG2_DRBG_state_handle,$
1723 $requested_number_of_bits, requested_security_strength, additional_input)$.

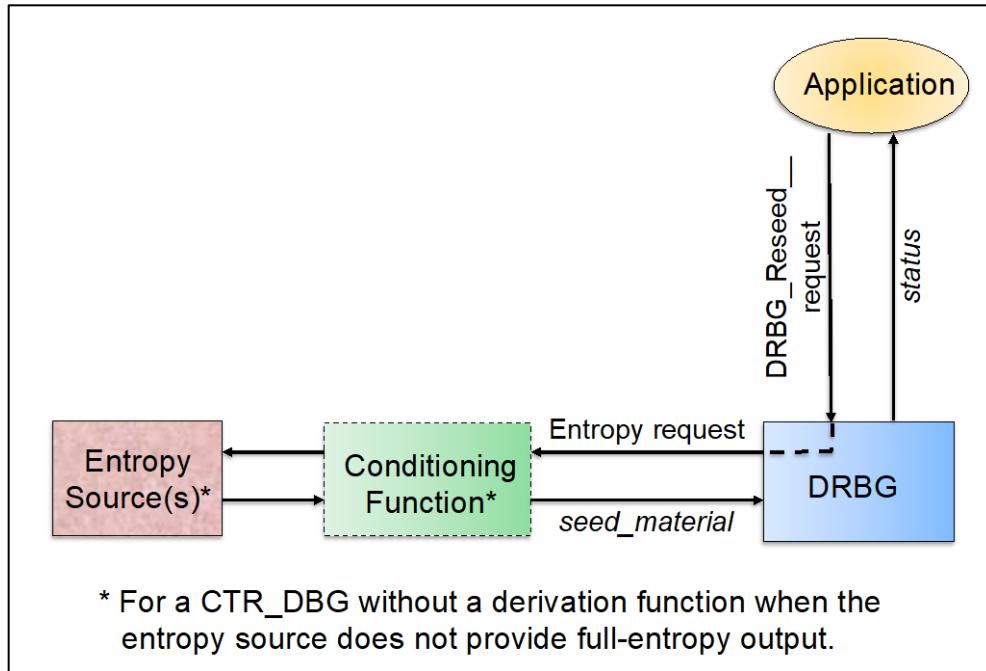
1724 The request **shall** result in the execution of a **DRBG_Generate** function by the DRBG (see Sec.
1725 2.8.1.2) and checking the *status* returned by the **DRBG_Generate** function:

- 1726 • $(status, returned_bits) = \text{DRBG_Generate}(RBG2_DRBG_state_handle,$
1727 $requested_number_of_bits, requested_security_strength, additional_input)$.
- 1728 • If $(status \neq \text{SUCCESS})$, then return $(status, Null)$.

1729 The **DRBG_Generate** function returns the *status* of the process, which **shall** also be returned to
1730 the application in response to the **DRBG_Generate_request**. If the *status* indicates that the
1731 generation was successful, the requested random bits (*returned_bits*) are also provided by the
1732 **DRBG_Generate** function and forwarded to the application.

1733 **5.2.3. Reseeding an RBG2 Construction**

1734 The capability to reseed an RBG2 construction is optional. If implemented, the reseeding of the
1735 DRBG may be performed 1) upon request from a consuming application or 2) based on
1736 implementation-selected criteria, such as time, number of outputs, events, or the availability of
1737 sufficient entropy. The DRBG **should** be reseeded occasionally (e.g., after 2^{19} bits have been
1738 output).



1739

Fig. 21. Reseed request from an application

1740 An application may request a reseed of the RBG2 construction (see Sec. 2.8.1.3):

1741 $status = \text{DRBG_Reseed_request}(RBG2_DRBG_state_handle, additional_input)$.

1742 If the DRBG receives a **DRBG_Reseed_Request** or if the DRBG is scheduled for a reseed (see SP
1743 800-90A), the **DRBG_Reseed** function **shall** be executed (see Sec. 2.8.1.3):

1744 $status = \text{DRBG_Reseed}(RBG2_DRBG_state_handle, additional_input)$.

1745 The **DRBG_Reseed** function returns the *status* of the reseed process, which **shall** be returned
1746 to the application if requested using a **DRBG_Reseed_request**.

1747 The **DRBG_Reseed** function uses a **Get_randomness-source_input** call to obtain the entropy
1748 needed for reseeding the DRBG (see Sec. 2.8.1.3 herein and SP 800-90A). The DRBG is reseeded
1749 at the instantiated security strength recorded in the DRBG's internal state. The
1750 **Get_randomness-source_input** call in SP 800-90A **shall** be replaced with the following:

1751 1. For the CTR_DRBG without a derivation function, use the appropriate replacement as
1752 specified in step 1 of Sec. 5.2.1.

- 1754 2. For CTR_DRBG (with a derivation function), Hash_DRBG, or HMAC_DRBG, replace
1755 the **Get_randomness-source_input** call in the **DRBG_Reseed** function with the
1756 following:¹⁸
- 1757 a. If the implementer wants full entropy in the returned bitstring, the
1758 **Get_randomness-source_input** call is replaced by:
- 1759 $(status, seed_material) = \text{Get_conditioned_full_entropy_input}(s, counting_method)$.
- 1761 b. Otherwise, the **Get_randomness-source_input** call is replaced by:
- 1762 $(status, seed_material) = \text{Get_entropy_bitstring}(s, counting_method)$
- 1763 OR
- 1764 $(status, seed_material) = \text{Get_conditioned_input}(s, counting_method)$.

1765 **5.3. RBG2 Construction Requirements**

- 1766 An RBG2 construction has the following requirements in addition to those specified in SP 800-
1767 90A and SP 800-90B:
- 1768 1. The RBG **shall** employ an **approved** and validated DRBG from SP 800-90A whose
1769 components are capable of providing the targeted security strength for the RBG.
- 1770 2. The RBG and its components **shall** be successfully validated for compliance with SP 800-
1771 90A, SP 800-90B, SP 800-90C, FIPS 140, and the specification of any other **approved**
1772 algorithm used within the RBG, as appropriate.
- 1773 3. One or more validated entropy sources **shall** be used to instantiate and reseed the DRBG.
1774 A non-validated entropy source **shall not** be used for this purpose.
- 1775 4. The DRBG **shall** be instantiated before first use (i.e., before providing output for use by a
1776 consuming application) and reseeded using the validated entropy source(s) used for
1777 instantiation (if a reseed capability is implemented).
- 1778 5. When instantiating and reseeding a CTR_DRBG without a derivation function, $s + 128$
1779 bits with full entropy **shall** be obtained either directly from the entropy source(s) or from
1780 the entropy source(s) via an external vetted conditioning function that provides full-
1781 entropy output (see Sec. 3.2.2.2).
- 1782 6. For CTR_DRBG (with a derivation function), Hash_DRBG, or HMAC_DRBG, a bitstring
1783 with at least $3s/2$ bits of entropy **shall** be obtained from the entropy source(s) to
1784 instantiate the DRBG at a security strength of s bits. When reseeding is performed, a
1785 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.

- 1786 entropy may be obtained directly from the entropy source(s) or via an external vetted
1787 conditioning function (see Sec. 3.2.2).
- 1788 7. The entropy source(s) used for the instantiation and reseeding of the DRBG within an
1789 RBG(P) construction **shall** include one or more validated physical entropy sources; the
1790 inclusion of one or more validated non-physical entropy sources is optional. A bitstring
1791 that contains entropy **shall** be assembled and the entropy in that bitstring determined as
1792 specified in Method 1 of Sec. 2.3 (i.e., only the entropy provided by validated physical
1793 entropy sources **shall** be counted toward fulfilling the amount of entropy in an entropy
1794 request).
- 1795 8. The entropy source(s) used for the instantiation and reseeding of the DRBG within an
1796 RBG2(NP) construction **shall** include one or more validated non-physical entropy sources;
1797 the inclusion of one or more validated physical entropy sources is optional. A bitstring
1798 containing entropy **shall** be assembled and the entropy in that bitstring determined as
1799 specified in Method 2 of Sec. 2.3 (i.e., the entropy provided by both validated non-
1800 physical entropy sources and any validated physical entropy sources included in the
1801 implementation **shall** be counted toward fulfilling the requested amount of entropy).
- 1802 9. A specific entropy-source output (or portion thereof) **shall not** be reused (e.g., it is
1803 destroyed after use).
- 1804 10. When a validated entropy source reports a failure, the failure **shall** be handled as
1805 discussed in item 10 of Sec. 2.6.

1806

1807 **6. RBG3 Constructions Based on the Use of Physical Entropy Sources**

1808 An RBG3 construction is designed to provide full entropy (i.e., an RBG3 construction can support
1809 all security strengths). An RBG3 construction is useful when bits with full entropy are required or
1810 a higher security strength than RBG1 and RBG2 constructions can support is needed.

1811 **6.1. General RBG3 Description**

1812 The RBG3 constructions specified in this recommendation include one or more physical entropy
1813 sources and an **approved** DRBG from SP 800-90A. One or more non-physical entropy sources may
1814 optionally be included, but any entropy they provide is not counted. That is, Method 1 of Sec. 2.3
1815 is used for counting entropy during RBG3 operation.

1816 Upon receipt of a request for random bits from a consuming application, the RBG3 construction
1817 accesses its entropy source(s) to obtain sufficient bits for the request. See Sec. 3.1 for further
1818 discussion about accessing entropy sources.

1819 An implementation may be designed so that the DRBG implementation used within an RBG3
1820 construction can be directly accessed by a consuming application using the same internal state
1821 as the RBG3 construction. Access to the DRBG using a different internal state than is used by the
1822 RBG3 construction is allowed as specified in Sec. 5 without the additional restrictions imposed in
1823 Sec. 6.3, Requirement 3, and Sec. 6.5.2, Requirements 2 and 3.

1824 The DRBG within an RBG3 construction is instantiated (i.e., seeded) at the highest security
1825 strength possible for its design (see Table 3). This is the fallback security strength if the entropy
1826 source fails in an undetected manner.

1827 **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

1828 If a failure of all physical entropy sources is detected, the RBG operation is terminated. Operation
1829 **must** be resumed only after repair and successful testing by instantiating the DRBG with new
1830 entropy from the entropy source(s).

1831 If all physical entropy sources fail in an undetected manner, the RBG continues to operate as an
1832 RBG2(P) construction, providing outputs at the security strength instantiated for its DRBG (see
1833 Sec. 5). Although security strengths of 128 and 192 bits are allowed for the DRBG (depending on
1834 its cryptographic primitive), a DRBG that is capable of supporting a security strength of 256 bits
1835 and is instantiated at that strength is recommended so that the RBG will continue to operate at

1836 a security strength of 256 bits in the event of an undetected failure of the physical entropy
1837 source(s).

1838 **6.2. RBG3 Construction Types and Their Variants**

1839 Two basic RBG3 constructions are specified:

- 1840 1. RBG3(XOR) — This construction is based on combining the output of one or more
1841 validated entropy sources with the output of an instantiated, **approved** DRBG using an
1842 exclusive-or operation (see Sec. 6.4).
- 1843 2. RBG3(RS) — This construction is based on using one or more validated entropy sources
1844 to continuously reseed the DRBG (see Sec. 6.5).

1845 **6.3. General Requirements**

1846 RBG3 constructions have the following general security requirements:

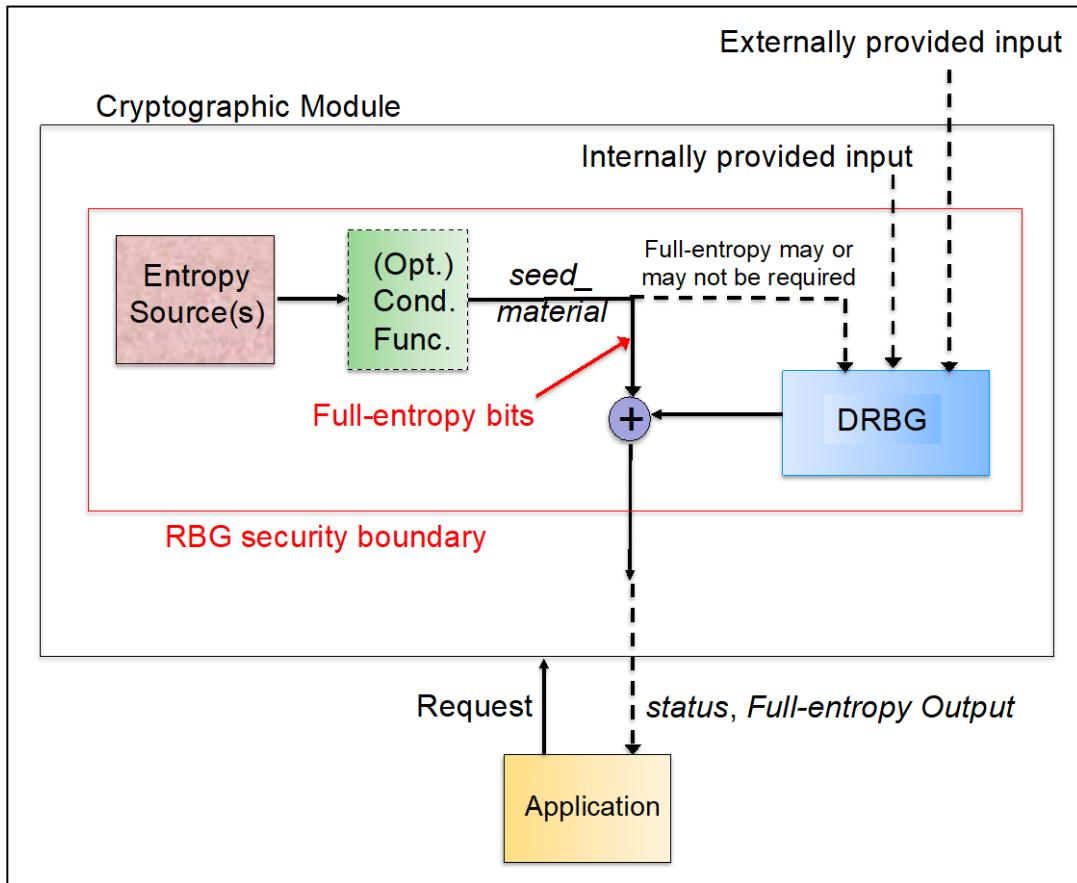
- 1847 1. An RBG3 construction **shall** be designed to provide outputs with full entropy using one or
1848 more validated, independent, physical entropy sources, as specified for Method 1 in Sec.
1849 2.3. Only the entropy provided by validated physical entropy sources **shall** be counted
1850 toward fulfilling entropy requests, although entropy provided by one or more validated
1851 non-physical entropy sources may be used but not counted.
- 1852 2. An RBG3 construction and its components **shall** be successfully validated for compliance
1853 with the corresponding requirements in SP 800-90A, SP 800-90B, SP 800-90C, FIPS 140,
1854 and the specification of any other **approved** algorithm used within the RBG, as
1855 appropriate.
- 1856 3. The DRBG **shall** be instantiated at its highest possible security strength before the first
1857 use of the RBG3 construction or direct access of the DRBG. A DRBG **should** support a
1858 security strength of 256 bits.
- 1859 4. The RBG **shall** employ an **approved** and validated DRBG from SP 800-90A whose highest
1860 possible security strength is the targeted fallback security strength for the DRBG (see Sec.
1861 6.1).
- 1862 5. A specific entropy-source output (or portion thereof) **shall not** be reused (e.g., the same
1863 entropy-source output **shall not** be used for an RBG3 request or for seeding or reseeding
1864 the DRBG).
- 1865 6. If the DRBG is directly accessible, the requirements in Sec. 5.3 for RBG2(P) constructions
1866 **shall** apply to the direct access of the DRBG.
- 1867 7. If a failure is detected within the RBG, see Sec. 2.6 (item 10) and 3.1.

1868 See Sec. 6.4.2 and 6.5.2 for additional requirements for the RBG3(XOR) and RBG3(RS)
1869 constructions, respectively.

1870 **6.4. RBG3(XOR) Construction**

1871 An RBG3(XOR) construction contains one or more validated entropy sources and a DRBG whose
 1872 outputs are XORed to produce full-entropy output during the generate process (see Fig. 22).

1873 In order to provide the required full-entropy output, the input to the XOR (shown as “ \oplus ” in the
 1874 figure) from the entropy-source side of the figure **shall** consist of bits with full entropy (see Sec.
 1875 2.1). If the entropy source(s) cannot provide full-entropy output, then an external conditioning
 1876 function **shall** be used to condition the output of the entropy source(s) to a full-entropy bitstring
 1877 before XORing with the output of the DRBG (see Sec. 3.2.2.2).



1878

1879 **Fig. 22. Generic structure of the RBG3(XOR) construction**

1880 When n bits of output are requested from an RBG3(XOR) construction, n bits of output from the
 1881 DRBG are XORed with n full-entropy bits obtained either directly from the entropy source(s) or
 1882 from the combination of validated entropy sources and an external vetted conditioning function
 1883 that provides full-entropy output (see Sec. 3.2.2.2). When the entropy sources are working
 1884 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.

1885 or to support a security strength of n bits. An example of an RBG3(XOR) design is provided in
1886 Appendix B.5.

1887 **6.4.1. Conceptual Interfaces**

1888 The RBG interfaces include function calls for instantiating the DRBG (see Sec. 6.4.1.1), generating
1889 random bits on request (see Sec. 6.4.1.2), and reseeding the DRBG instantiation (see Sec. 6.4.1.3).

1890 **6.4.1.1. Instantiation of the DRBG**

1891 As discussed in Sec. 2.8.3.1, before the RBG3(XOR) construction can be used to generate bits, an
1892 application instantiates the DRBG within the construction:

1893 $(status, state_handle) = \text{Instantiate_RBG3_DRBG_request}(requested_security_strength,$
1894 $personalization_string),$

1895 where $requested_security_strength$ and $personalization_string$ are optional. If the
1896 $requested_security_strength$ parameter is provided and exceeds the highest security strength
1897 that can be supported by the DRBG, an error indication **shall** be returned with an invalid
1898 $state_handle$ (see Sec. 2.8.3.1).

1899 If the $requested_security_strength$ is provided and is acceptable (i.e., $requested_security_strength$
1900 does not exceed the highest security strength that can be supported by the DRBG; see Sec.
1901 2.8.3.1) or if the $requested_security_strength$ parameter is not provided, the
1902 **Instantiate_RBG3_DRBG_request** received by the RBG3(XOR) construction **shall** result in the
1903 execution of the **RBG3(XOR)_Instantiate** function below. The $status$ returned by the
1904 **RBG3(XOR)_Instantiate** function **shall** be returned to the application in response to the
1905 **Instantiate_RBG3_DRBG_request**. The return of the $state_handle$ is optional if only a single
1906 instantiation is allowed by an implementation.

1907 Let s be the highest security strength that can be supported by the DRBG. The DRBG in the
1908 RBG3(XOR) construction is instantiated as follows:

1909 **RBG3(XOR)_Instantiate:**

1910 **Input:**

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

1913 **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
1916 DRBG or an invalid state handle.

1917 **Process:**

- 1918 1. $(status, RBG3_DRBG_state_handle) = \mathbf{DRBG_Instantiate}(s,$
1919 *personalization_string*).
1920 2. If $(status \neq \text{SUCCESS})$, then return $(status, Invalid_state_handle)$.
1921 3. Return $(\text{SUCCESS}, RBG3_DRBG_state_handle)$.
- 1922 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.
- 1925 In step 2, if the $status$ returned from step 1 does not indicate a success, then return the $status$ with an invalid state handle.
- 1927 In step 3, the $status$ and $RBG3_DRBG_state_handle$ that were obtained in step 1 are returned to the requesting application.
- 1929 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.

1930 **6.4.1.2. Random Bit Generation by the RBG3(XOR) Construction**

1931 As discussed in Sec. 2.8.3.2, an application may request the generation of random bits from the
1932 RBG3(XOR) construction:

1933 $(status, returned_bits) = \mathbf{RBG3_DRBG_Generate_request}(RBG3_DRBG_state_handle, n,$
1934 *additional_input*),

1935 where $RBG3_DRBG_state_handle$ was provided during instantiation (see Sec. 6.4.1.1), n is the
1936 number of bits to be generated and returned to the application, and *additional_input* is optional.

1937 The **RBG3_DRBG_Generate_request** received by the RBG3(XOR) construction **shall** result in
1938 the execution of the **RBG3(XOR)_Generate** function below. The output of that function **shall**
1939 be returned to the application in response to the **RBG3_DRBG_Generate_request**.

1940 Let s be the security strength instantiated for the DRBG (i.e., the highest security strength that
1941 can be supported by the DRBG; see Sec. 6.4.1.1), and let the $RBG3_DRBG_state_handle$ be the
1942 value returned by the instantiation function for RBG3(XOR)'s DRBG instantiation. Random bits
1943 with full entropy **shall** be generated by the RBG3(XOR) construction using the following generate
1944 function with the values of n and *additional_input* provided in the **DRBG_Generate_request** as
1945 input:

1946 **RBG3(XOR)_Generate:**

1947 **Input:**

- 1948 1. $RBG3_DRBG_state_handle$: The state handle of the DRBG used by the RBG3
1949 construction.
1950 2. n : The number of bits to be generated.

1951 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. $(status, ES_bits) = \text{Request_entropy}(n)$. (See the notes below for customizing this step.)

2. If ($status \neq \text{SUCCESS}$), then return ($status, Null$).

3. $(status, DRBG_bits) = \text{DRBG_Generate}(RBG3_DRBG_state_handle, n, s, additional\ input).$

4. If ($status \neq \text{SUCCESS}$), then return ($status, Null$).

5. *returned bits* = *ES bits* \oplus *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:

$(status, ES\ bits) = \text{Get_entropy_bitstring}(n, Method\ 1)$.

The **Get_entropy_bitstring** function²⁰ 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:

$(status, ES\ bits) = \text{Get_conditioned_full_entropy_input}(n, Method\ I).$

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).

1981 In step 2, if the request in step 1 is not successful, abort the **RBG3(XOR)_Generate** function,
1982 returning the *status* received in step 1 and a *Null* bitstring as the *returned_bits*. If *status* indicates
1983 a success, *ES_bits* is the full-entropy bitstring to be used in step 5.

²⁰ See Sec. 2.8.2 and 3.2.

1984 In step 3, the RBG3(XOR)'s DRBG instantiation is requested to generate n bits at a security
1985 strength of s bits. The DRBG instantiation is indicated by the *RBG3_DRBG_state_handle*, which
1986 was obtained during instantiation (see Sec. 6.4.1.1). If additional input is provided in the
1987 **RBG3(XOR)_Generate** call, it **shall** be included in the **DRBG_Generate** function call to the
1988 DRBG. It is possible that the DRBG may require reseeding during the **DRBG_Generate** function
1989 call in step 3 (e.g., because the end of the seedlife of the DRBG has been reached).

1990 In step 4, if the **DRBG_Generate** function request is not successful, the **RBG3(XOR)_Generate**
1991 function is aborted, and the *status* received in step 3 and a *Null* bitstring are returned to the
1992 consuming application. If *status* indicates a success, *DRBG_bits* is the pseudorandom bitstring to
1993 be used in step 5.

1994 Step 5 combines the bitstrings returned from the entropy source(s) (from step 1) and the DRBG
1995 (from step 3) using an XOR operation. The resulting bitstring is returned to the consuming
1996 application in step 6.

1997 6.4.1.3. Pseudorandom Bit Generation Using a Directly Accessible DRBG

1998 If prediction resistance is desired by a consuming application for the next DRBG output to be
1999 generated so that a previous internal state that may have been compromised cannot be used to
2000 determine the next DRBG output, the application requests a reseed of the DRBG before
2001 requesting the generation of pseudorandom bits directly from the DRBG, as discussed in Sec.
2002 6.4.1.4. This is the same process shown in Fig. 20 in Sec. 5.2.2.

2003 If a reseed of the DRBG was not requested by the application, or a *status* of SUCCESS was returned
2004 by the **DRBG_Reseed** function when the application requested a reseed, pseudorandom bits
2005 may be requested as follows:

2006 $(status, returned_bits) = \text{DRBG_Generate_request}(RBG3(XOR)_DRBG_state_handle,$
2007 $requested_number_of_bits, requested_security_strength, additional_input),$

2008 where *RBG3(XOR)_state_handle* was provided during instantiation and *additional_input* is
2009 optional.

2010 The **DRBG_Generate_request** received by the DRBG **shall** result in the execution of the
2011 **DRBG_Generate** function in the DRBG:

2012 $(status, returned_bits) = \text{DRBG_Generate}(RBG3_DRBG_state_handle,$
2013 $requested_number_of_bits, requested_security_strength, additional_input),$

2014 where:

- 2015 • *RBG3_DRBG_state_handle* is the state handle used by the DRBG within the RBG3(XOR)
2016 construction.
- 2017 • *requested_security_strength* is provided in the **DRBG_Generate_request** and must be \leq
2018 the instantiated security strength of the DRBG.

- 2019 • Any *additional_input* provided in a **DRBG_Generate_request** shall be provided as input
2020 to the **DRBG_Generate** function. Otherwise, the use of *additional_input* is optional.

2021 The output of the **DRBG_Generate** function shall be returned to the application in response to
2022 the **DRBG_Generate_request**.

2023 **6.4.1.4. Reseeding the DRBG Instantiation**

2024 As discussed in Sec. 2.4.2, the reseeding of the DRBG may be performed 1) upon request from a
2025 consuming application or 2) based on implementation-selected criteria, such as time, number of
2026 outputs, events, or the availability of sufficient entropy.

2027 An application may request the reseeding of the DRBG within the RBG3(XOR) construction:

2028 $status = \text{DRBG_Reseed_request}(RBG3(\text{XOR})_DRBG_state_handle, additional_input)$,

2029 where *RBG3(XOR)_state_handle* was provided during instantiation and *additional_input* is
2030 optional.

2031 The DRBG executes a **DRBG_Reseed** function in response to a **DRBG_Reseed_request** from an
2032 application or in accordance with implementation-selected criteria:

2033 $status = \text{DRBG_Reseed}(RBG3_DRBG_state_handle, additional_input)$,

2034 where *RBG3_DRBG_state_handle* (if used) was returned by the **DRBG_Instantiate** function
2035 (see Sec. 2.8.1.1 and 6.4.1.1). *RBG3_DRBG_state_handle* is the state handle for the internal state
2036 of the DRBG within the RBG3(XOR) construction. Any *additional_input* provided in a
2037 **DRBG_Reseed_request** shall be provided as input to the **DRBG_Reseed** function. Otherwise,
2038 the use of *additional_input* is optional.

2039 **6.4.2. RBG3(XOR) Requirements**

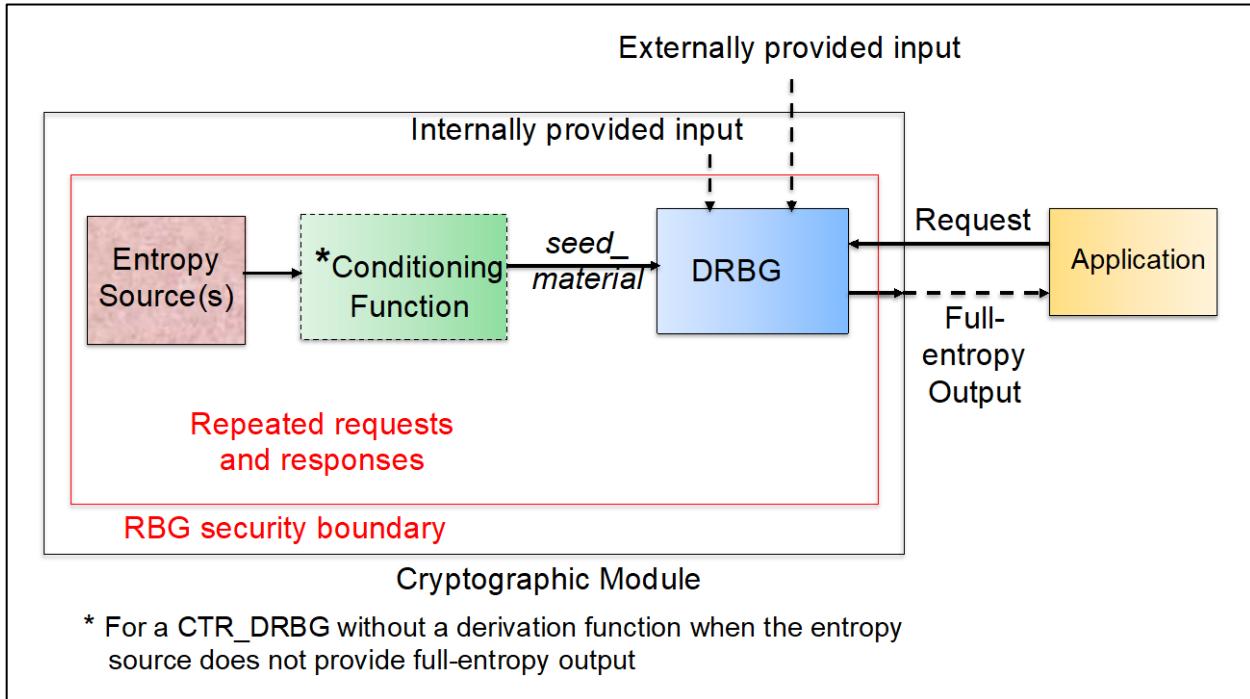
2040 An RBG3(XOR) construction has the following requirements in addition to those provided in Sec.
2041 6.3:

- 2042 1. Bitstrings with full entropy shall be provided to the XOR operation either directly from
2043 the concatenated output of one or more validated physical entropy sources or by an
2044 external conditioning function that provides full-entropy output using the output of one
2045 or more validated physical entropy sources.
- 2046 2. Entropy source output used for the RBG's XOR operation shall not also be used to
2047 initialize and reseed the RBG's DRBG.²¹
- 2048 3. The DRBG instantiation should be reseeded occasionally (e.g., after a predetermined
2049 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.

2050 6.5. RBG3(RS) Construction

2051 The second RBG3 construction specified in this document is the RBG3(RS) construction shown in
2052 Fig. 23. An example of this construction is provided in Appendix B.6.



2053 2054 **Fig. 23. Generic structure of the RBG3(RS) construction**

2055 External conditioning of the outputs from the entropy source(s) during instantiation and
2056 reseeding is required to provide bitstrings with full entropy when the DRBG is a CTR_DRBG
2057 without a derivation function and the entropy source(s) do not provide output with full entropy.
2058 Otherwise, the use of a conditioning function is optional.

2059 6.5.1. Conceptual Interfaces

2060 The RBG interfaces include function calls for instantiating the DRBG (see Sec. 6.5.1.1), generating
2061 random bits on request (see Sec. 6.5.1.2), and reseeding the DRBG instantiation (see Sec. 6.5.1.3).

2062 6.5.1.1. Instantiation of the DRBG Within an RBG3(RS) Construction

2063 Before the RBG3(RS) construction can be used to generate bits, an application **shall** request the
2064 instantiation of the DRBG within the construction (see Sec. 2.8.3.1):

2065 $(status, RBG3_DRBG_state_handle) =$

2066 **Instantiate_RBG3_DRBG_request**(*requested_security_strength*, *personalization_string*),

2067 where *requested_security_strength* and *personalization_string* are optional. If the
2068 *requested_security_strength* parameter is provided and exceeds the highest security strength

2069 that can be supported by the DRBG design, an error indication **shall** be returned with an invalid
2070 *state_handle* (see Sec. 2.8.3.1).

2071 If the *requested_security_strength* is provided and acceptable (see Sec. 2.8.3.1) or the
2072 *requested_security_strength* information is not provided, the
2073 **Instantiate_RBG3_DRBG_request** received by the RBG3(RS) construction **shall** result in the
2074 execution of the **RBG3(RS)_Instantiate** function below. The *status* returned by that function
2075 **shall** be returned to the application in response to the **Instantiate_RBG3_DRBG_request**.

2076 Let *s* be the highest security strength that can be supported by the DRBG, and let
2077 *personalization_string* be the value provided in the **Instantiate_RBG3_DRBG_request** (if any).
2078 The DRBG in the RBG3(RS) construction is instantiated as follows:

2079 **RBG3(RS)_Instantiate:**

2080 **Input:**

- 2081 1. *s*: The requested security strength for the DRBG in the RBG3(RS) construction.
2082 2. *personalization_string*: An optional (but recommended) personalization string.

2083 **Output:**

- 2084 1. *status*: The status returned from the **RBG3(RS)_Instantiate** function.
2085 2. *RBG3_DRBG_state_handle*: A pointer to the internal state of the DRBG if the *status*
2086 indicates a success. Otherwise, an invalid state handle.

2087 **Process:**

- 2088 1. (*status*, *RBG3_DRBG_state_handle*) = **DRBG_Instantiate**(*s*,
2089 *personalization_string*).
2090 2. If (*status* ≠ SUCCESS), then return (*status*, *Invalid_state_handle*).
2091 3. Return (SUCCESS, *RBG3_DRBG_state_handle*).

2092 In step 1, the DRBG is instantiated at a security strength of *s* bits.

2093 In step 2, if the *status* returned from step 1 does not indicate a success, then return the *status*
2094 and an invalid state handle.

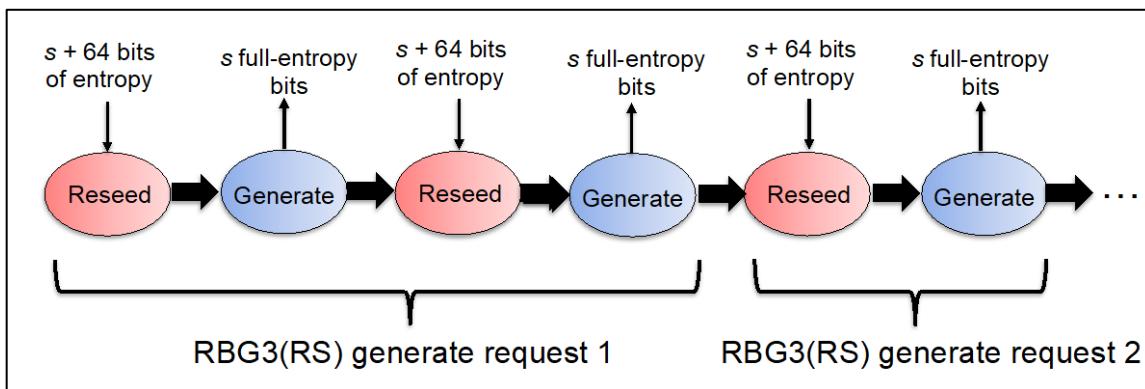
2095 In step 3, the *status* and the *RBG3_DRBG_state_handle* are returned.
2096 *RBG3_DRBG_state_handle* is the state handle for the internal state of the DRBG used within the
2097 RBG3(RS) construction.

2098 The handling of status codes is discussed in Sec. 2.8.3 and 6.5.1.2.

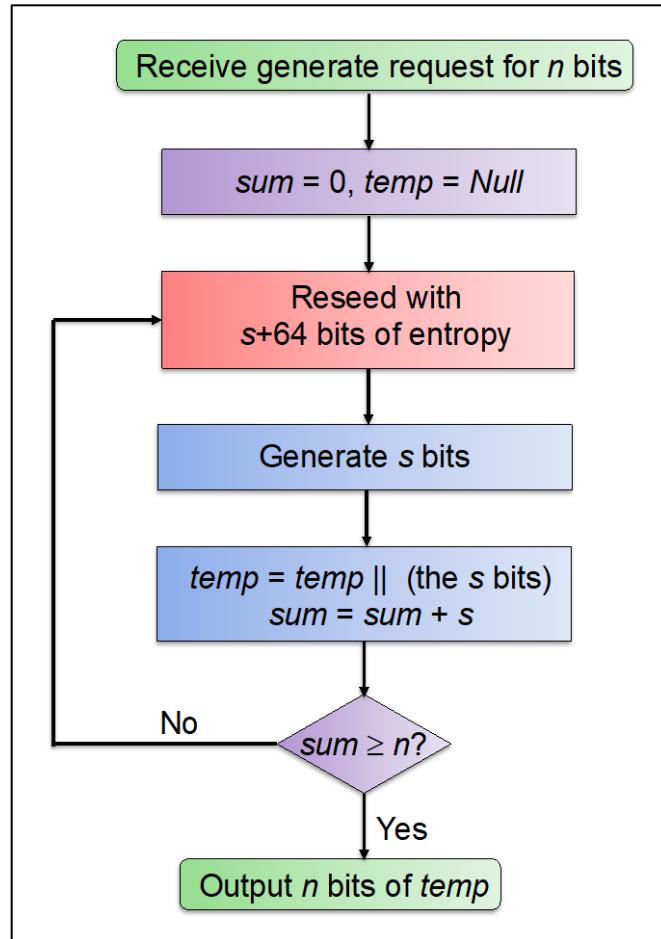
2099 **6.5.1.2. Random and Pseudorandom Bit Generation**2100 **6.5.1.2.1. Generation Using the RBG3(RS) Construction**

2101 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.

2108 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).

2118 **Fig. 24. Sequence of RBG3(RS) generate requests**

2119 Figure 25 provides a flow of the steps of the **RBG3(RS)_Generate** function.



2120

2121

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

2122 Figure 26 depicts a sequence of RBG3(RS) generate requests followed by a sequence of requests
 2123 directly to the DRBG (shown in green) and another sequence of RBG3(RS) generate requests. As
 2124 previously discussed, an RBG3(RS) generate request is preceded by obtaining $s + 64$ bits of fresh
 2125 entropy. The first generate request directly to the DRBG following one or more RBG3(RS)
 2126 generate requests is preceded by obtaining $s + 64$ bits of fresh entropy. Successive DRBG requests
 2127 do not require the insertion of fresh entropy (except, for example, if requested by the consuming
 2128 application). When a consuming application later requests that the RBG3(RS) construction
 2129 generate full-entropy bits again, the reseed-generate process is resumed by first reseeding with
 2130 $s + 64$ bits of entropy before the generation of each s -bit string by the RBG3(RS) construction.

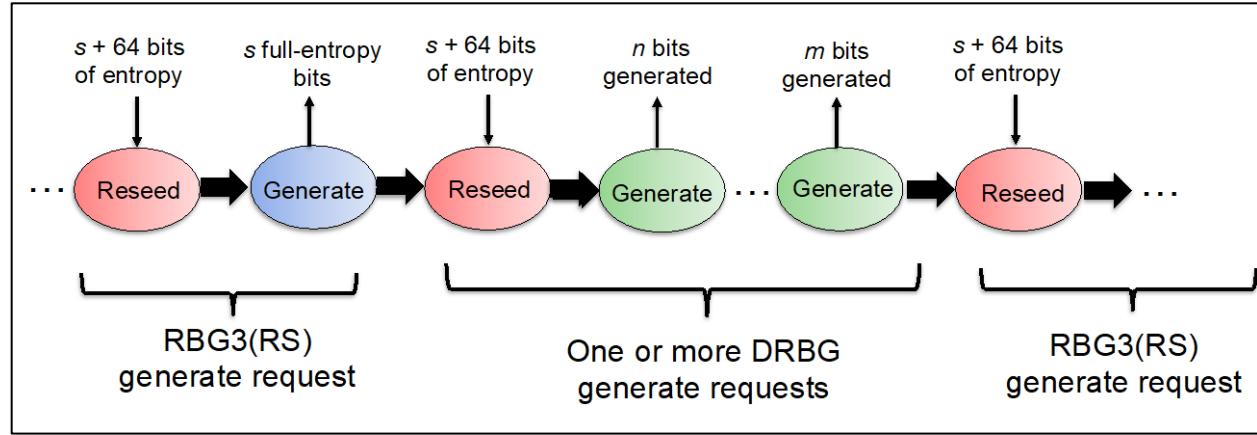


Fig. 26. Direct DRBG generate requests

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`.

2156 2. $sum = 0$.
2157 3. While ($sum < n$),
2158 3.1 Reseed with at least $s + 64$ bits of fresh entropy (see the notes below for
2159 customizing this step).
2160 3.2 $(status, full_entropy_bits) = \mathbf{DRBG_Generate}(RBG3_DRBG_state_handle, s,$
2161 $s, additional_input)$.
2162 3.3 If ($status \neq \text{SUCCESS}$), then return ($status, Null$).
2163 3.4 $temp = temp \parallel full_entropy_bits$.
2164 3.5 $sum = sum + s$.
2165 3.6 $additional_input = Null$ string.
2166 4. Return ($\text{SUCCESS}, \mathbf{leftmost}(temp, n)$).

2167 In steps 1 and 2, the bitstring intended to collect the generated bits ($temp$) is initialized to the
2168 *Null* bitstring, and the counter for the number of bits obtained for fulfilling the request (sum) is
2169 initialized to zero.

2170 Step 3 is iterated until at least n full-entropy bits have been generated.

2171 Step 3.1 obtains at least $s + 64$ bits of fresh entropy and inserts it into the internal state.

- 2172 • For CTR_DRBG without a derivation function, $s + 128$ bits of entropy are requested
2173 during reseeding using a randomness source that provides full-entropy output. Step 3.1
2174 becomes:

2175 ○ $status = \mathbf{DRBG_Reseed}(RBG3_DRBG_state_handle, additional_input)$.
2176 ○ If ($status \neq \text{SUCCESS}$), then return ($status, Null$)
2177 with the **Get_randomness-source_input** call in the **DRBG_Reseed** function replaced
2178 by:
2179 ○ $(status, seed_material) = \mathbf{Get_entropy_bitstring}(s + 128, Method_I)$.
2180 ○ If ($status \neq \text{SUCCESS}$), then return ($status, Null$),
2181 where *Method_I* indicates that only the entropy from physical entropy sources is
2182 counted.

- 2183 • For a Hash_DRBG, HMAC_DRBG, or CTR_DRBG with a derivation function, s bits of
2184 fresh entropy are usually inserted into the internal state during a **DRBG_Reseed**
2185 function. To insert $s + 64$ bits into the internal state, two methods are provided:

2186 Method A is a modification of the **DRBG_Reseed** function that requests $s + 64$ bits of
2187 entropy from the entropy source(s) rather than (the usual) s bits (see Fig. 27). Making this
2188 change is straightforward, given access to the internals of a DRBG implementation.

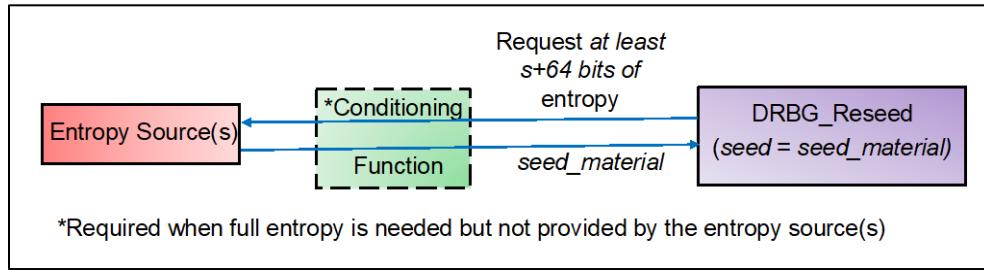


Fig. 27. Modification of the DRBG_Reseed function

Step 3.1 becomes:

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

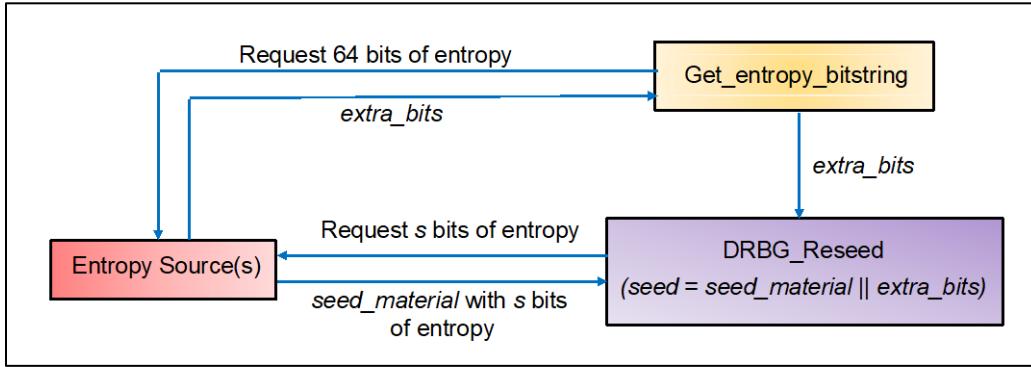
with the `Get_randomness-source_input` call in the **DRBG_Reseed** function replaced by:

- `(status, seed_material) = Get_entropy_bitstring($s + 64$, Method_I).`
- If (`status ≠ SUCCESS`), then return (`status, Null`).

Method_I 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).



2209

Fig. 28. Request extra bits before reseeding

2210 Step 3.1 becomes:

- 2211 3.1.1 $(status, extra_bits) = \text{Get_entropy_bitstring}(64, Method_I)$.
- 2212 3.1.2 If $(status \neq \text{SUCCESS})$, then return $(status, Null)$.
- 2213 3.1.3 $status = \text{DRBG_Reseed}(RBG3_DRBG_state_handle, extra_bits \parallel additional_input)$.
- 2214
- 2215 3.1.4 If $(status \neq \text{SUCCESS})$, then return $(status, Null)$.

2216 In step 3.1.3, the **Get_randomness-source_input** call in the **DRBG_Reseed** function is replaced by:

- 2217 o $(status, seed_material) = \text{Get_entropy_bitstring}(s, Method_I)$.
- 2218 o If $(status \neq \text{SUCCESS})$, then return $(status, Null)$.

2219 *Method_I* indicates that only the entropy from physical entropy sources is to be counted.

2220 In step 3.2, request the generation of *full_entropy_bits* using the **DRBG_Generate** function,
2221 where:

- 2222 • The *RBG3_DRBG_state_handle* was obtained during DRBG instantiation (see Sec.
2223 6.5.1.1).
- 2224 • *s* is both the number of full-entropy bits to be produced during the **DRBG_Generate**
2225 function call and the security strength of the DRBG instantiation (see Sec. 2.8.1.2 and
2226 Table 4 in Appendix A.2).
- 2227 • *additional_input* is the current value of the *additional_input* string (initially provided in
2228 the **DRBG_Generate** call, used in the first iteration of step 3.2, and subsequently set to
2229 the *Null* string in step 3.6).

2230 In step 3.3, if step 3.2 returned a *status* value indicating that the **DRBG_Generate** function
2231 was not successful, then return the *status* to the calling application with a *Null* bitstring.

2232 In step 3.4, concatenate the *full_entropy_bits* obtained in step 3.2 to the temporary bitstring
2233 (*temp*).

2236 In step 3.5, increment the output-length counter (*sum*) by *s* bits (i.e., the number of full-
2237 entropy bits obtained in step 3.2).

2238 In step 3.6, to avoid reusing the *additional_input*, set its value to a *Null* string for subsequent
2239 iterations of step 3.

2240 If *sum* < *n*, go to step 3.1.

2241 Step 4 returns a *status* indicating SUCCESS to the calling application along with the leftmost *n* bits
2242 of *temp* as the *returned_bitstring*.

2243 **6.5.1.2.2. Generation Using a Directly Accessible DRBG**

2244 As discussed in Sec. 2.8.1.2, the DRBG used by the RBG3(RS) construction may be requested to
2245 generate output directly using the following request:

2246 $(status, returned_bits) = \text{DRBG_Generate_request}(RBG3_DRBG_state_handle,$
2247 $requested_number_of_bits, requested_security_strength, additional_input),$

2248 where *RBG3_DRBG_state_handle* was provided during instantiation (see Sec. 6.5.1.1) and
2249 *additional_input* is optional.

2250 Before generating the requested output, the DRBG needs to be reseeded in the following
2251 circumstances:

- 2252 1. Accessing a DRBG directly to generate output by the DRBG in the RBG3(RS) construction
2253 requires that the DRBG be reseeded with at least *s* + 64 bits of entropy from the entropy
2254 source(s) when the DRBG was previously used as a component of the
2255 **RBG3(RS)_generate** function. This requires that the RBG3(RS) implementation keep
2256 track of the type of generate request that was made previously (e.g., including this
2257 information in the DRBG's internal state) so that the reseeding of the DRBG is
2258 automatically performed before generating the requested DRBG output.
- 2259 2. During a sequence of generate requests, the DRBG may reseed itself in response to some
2260 event.

2261 Reseeding is accomplished as specified in Sec. 6.5.1.3.

2262 If a reseed of the DRBG was not performed or a *status* of SUCCESS was returned by the
2263 **DRBG_Reseed** function when performed under conditions 1 or 2 above, the
2264 **DRBG_Generate_request** invokes the **DRBG_Generate** function (see Sec. 5.2.2), obtains the
2265 *status* of the operation and any generated bits (i.e., *returned_bits*), and forwards them to the
2266 application in response to the **DRBG_Generate_request**.

2267 **6.5.1.3. Reseeding**

2268 Reseeding the DRBG may be performed:

- 2269 1. When explicitly requested by the consuming application,

- 2270 2. During an **RBG3(RS)_generate** request (see Sec. 6.5.1.2.1) or in response to a direct
2271 DRBG generate request when the previous use of the DRBG was as a component of the
2272 **RBG3(RS)_Generate** function (see Sec. 6.5.1.2.2), or
2273 3. Based on implementation-selected criteria, such as time, number of outputs, events, or
2274 the availability of sufficient entropy.

2275 **Case 1:** An application sends a reseed request to the RBG:

2276 *status* = **DRBG_Reseed_request**(*RBG3_DRBG_state_handle*, *additional_input*),
2277 where *RBG3_DRBG_state_handle* was obtained during instantiation (see Sec. 6.5.1.1) and
2278 *additional_input* is optional.

2279 Any *additional_input* provided by a **DRBG_Reseed request** from the application **shall** be
2280 used as input to the **DRBG_Reseed** function. Otherwise, the use of *additional_input* is
2281 optional.

2282 The **DRBG_Reseed_request** results in the invocation of the **DRBG_Reseed** function (see
2283 Sec. 5.2.3). The *status* returned from the **DRBG_Reseed** function is forwarded to the
2284 application in response to the **DRBG_Reseed_request**.

2285 **Case 2:** The DRBG is reseeded as follows:

- 2286 • For CTR_DRBG without a derivation function, $s + 128$ bits of entropy are requested
2287 during reseeding in the same manner as for instantiation (see step 3.1 of Sec. 6.5.1.2.1).
2288 • For a Hash_DRBG, HMAC_DRBG, or CTR_DRBG with a derivation function, use
2289 Method A or Method B (as specified in step 3.1 of Sec. 6.5.1.2.1) to obtain $s + 64$ bits of
2290 fresh entropy in the DRBG.

2291 **Case 3:** A reseed of the DRBG is invoked based on implementation-selected criteria:

2292 *status* = **DRBG_Reseed**(*RBG3_DRBG_state_handle*, *additional_input*).

2293 For a CTR_DRBG, the DRBG is reseeded with $s + 128$ bits of fresh entropy. Otherwise, the
2294 DRBG is reseeded with either s or $s + 64$ bits of fresh entropy, depending on whether Method
2295 A or Method B was used in step 3.1 of Sec. 6.5.1.2.1.

2296 **6.5.2. Requirements for an RBG3(RS) Construction**

2297 An RBG3(RS) construction has the following requirements in addition to those provided in Sec.
2298 6.3:

- 2299 1. For each s bits generated by the RBG3(RS) construction, $s + 64$ bits of fresh entropy **shall**
2300 be acquired either directly from independent, validated entropy sources or from an
2301 external conditioning function that processes the output of the validated entropy sources
2302 to provide full-entropy, as specified in Sec. 3.2.2.2.

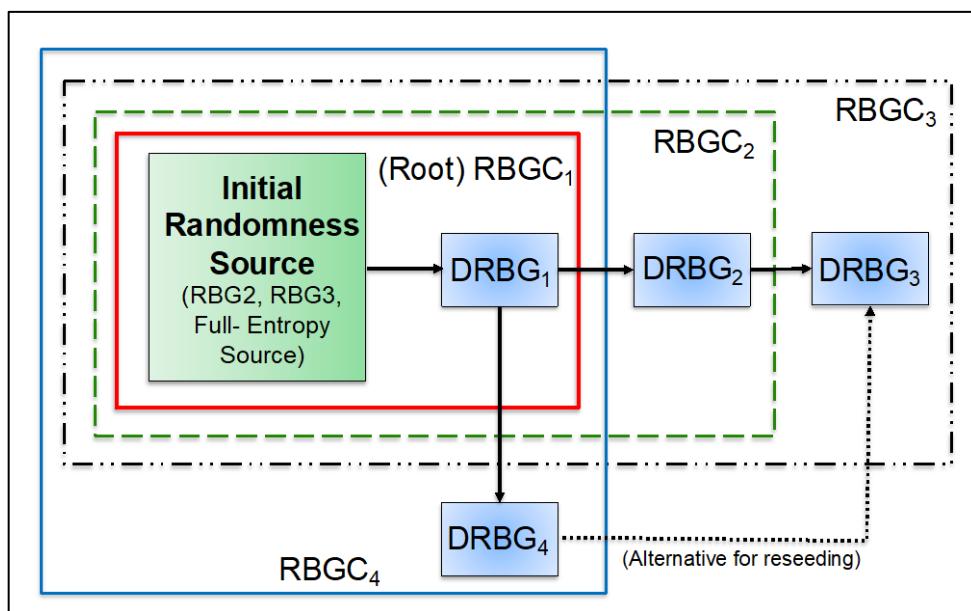
- 2303 2. If the DRBG is directly accessible and the previous use of the DRBG was by the RBG3(RS)
2304 construction, a reseed of the DRBG instantiation with at least $s + 64$ bits of entropy **shall**
2305 be performed before generating output.
- 2306 3. The DRBG **shall** be reseeded in accordance with Sec. 6.5.1.3.
- 2307

2308 **7. RBGC Construction for DRBG Chains**

2309 The RBGC construction allows the use of a chain of DRBGs in which one DRBG is used to provide
2310 seed material for another DRBG. This design is common on many computing platforms and allows
2311 some level of modularity (e.g., an operating system RBG can be designed and validated without
2312 knowing the randomness source that will be available on the particular hardware on which it will
2313 be used, or a software application can be designed with its own RBG but without knowing the
2314 operating system or hardware used by the application).

2315 **7.1. RBGC Description**2316 **7.1.1. RBGC Environment**

2317 Figure 29 depicts RBGC constructions and the environment in which they will be used. An RBGC
2318 construction consists of an **approved** DRBG mechanism (from SP 800-90A) and the randomness
2319 source used for seeding and (optional) reseeding. This figure illustrates a tree of RBGC
2320 constructions that consists of two DRBG chains: 1) a chain consisting of DRBG₁, DRBG₂, and DRBG₃
2321 and 2) a chain consisting of DRBG₁ and DRBG₄.

2322
2323 **Fig. 29. DRBG tree using the RBGC construction**

2324 The core of this type of construction is called the *root* and is shown as RBGC₁ within the solid red
2325 rectangle in the figure. Its DRBG is labeled as DRBG₁, and its randomness source for seeding and
2326 (optionally) reseeding is labeled as the *initial randomness source*.

2327 For each of the other RBGC constructions (i.e., RBG₂, RBG₃, and RBG₄), the DRBG within the
2328 construction is seeded by a DRBG within a “parent” RBGC construction (i.e., the parent is the
2329 randomness source used for seeding the DRBG). For RBGC₂ (shown as a box outlined with long

2330 green dashes [— — —]), the parent randomness source is the root (i.e., RBGC₁). For RBGC₃ (shown
2331 as a box with black dashes and dots [— • — • —]), the parent randomness source is RBGC₂. For
2332 RBGC₄ (shown within a box outlined with a solid blue rectangle), the parent randomness source
2333 is RBGC₁ (i.e., the root).

2334 An RBGC construction may be used to Instantiate and reseed other RBGC constructions or to
2335 provide output for one or more applications (not shown in Fig. 29). All components of an RBGC
2336 tree — including the initial randomness source and the DRBG chains in that tree — reside on the
2337 same computing platform. The initial randomness source is not physically removable while the
2338 computing platform is operational, and the contents of the internal state of any DRBG in the tree
2339 are never relocated to another computer platform or output for external storage. See Appendix
2340 A.3 for a discussion about the intended meaning of a computing platform and implementation
2341 considerations.

2342 Each RBGC construction may be a *parent* for one or more *child* RBGC constructions. Each of the
2343 child RBGC constructions has only one parent that serves as its randomness source for seeding
2344 the DRBG within it. Using Fig. 29 as an example, RBGC₁ is the only parent of both RBGC₂ and
2345 RBGC₄. RBGC₂ is the randomness source (i.e., the only parent) of RBGC₃. However, the parent
2346 may have siblings that may be used for reseeding under certain conditions (see Sec. 7.1.2.1) if
2347 the parent is not available to do so (e.g., the RBGC construction has been moved to a different
2348 core). In Fig. 29, RBGC₂ and RBGC₄ are siblings since they have the same parent (RBGC₁). In this
2349 case, the alternative path for reseeding is shown as a line of black dots.

2350 An RBGC construction cannot have itself as a predecessor randomness source for reseeding. That
2351 is, there are no “seed loops” in which an RBGC construction provides seed material for a
2352 predecessor RBGC construction (e.g., a parent or grandparent). For example, in Fig. 29, RBGC₂
2353 can be used as the randomness source for RBGC₃, but RBGC₃ cannot be used as the randomness
2354 source for reseeding RBGC₁ or RBGC₂. However, *additional_input* provided to the DRBG during a
2355 reseed or generate request may be anything, including the output of any RBGC construction of
2356 the tree.

2357 **7.1.2. Instantiating and Reseeding Strategy**

2358 **7.1.2.1. Instantiating and Reseeding the Root RBGC Construction**

2359 The root RBGC construction is instantiated and (optionally) reseeded using an initial randomness
2360 source, which is either a validated full-entropy source or a validated RBG2(P), RBG2(NP),
2361 RBG3(XOR), or RBG3(RS) construction. An RBG2(P) or RBG2(NP) construction used as the initial
2362 randomness source **shall** have a capability of being reseeded on demand by the root.²³ A
2363 validated full-entropy source is a validated entropy source that provides full-entropy output or
2364 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.

2365 provides full-entropy output (see Sec. 3.2.2.2). The root may provide prediction resistance if
2366 reseeded by the initial random source.

2367 **7.1.2.2. Instantiating and Reseeding a Non-Root RBGC Construction**

2368 Each non-root RBGC construction in a chain is instantiated by a single RBGC construction (i.e., its
2369 parent) using that parent as its randomness source. If the child RBGC construction can be
2370 reseeded, the parent normally serves as the randomness source during the reseeding process.
2371 However, if the parent is not available for reseeding (e.g., the implementation of the RBGC
2372 construction has been moved to a different core on the computing platform), a sibling of the
2373 parent may be used as an alternative randomness source provided that:

- 2374 1. The sibling has been validated for compliance with an RBGC construction, and
2375 2. The DRBG within the sibling supports the security strength of the DRBG to be reseeded.

2376 Using Fig. 29, consider RBGC₃ as the target RBGC construction to be reseeded. RBGC₂ is the parent
2377 of RBGC₃ and would normally be used as the randomness source for reseeding RBGC₃. If RBGC₂
2378 is not available when RBGC₃ needs to be reseeded, then a sibling of RBGC₂ may be used as an
2379 alternative randomness source for reseeding if it meets conditions 1 and 2 above. In Fig. 29,
2380 RBGC₄ is depicted as a sibling of RBGC₂, so RBGC₄ may be used as an alternative randomness
2381 source (as indicated by the path of black dots) if it is validated for that purpose and the DRBG
2382 within the RBGC₄ construction can support the security strength of RBGC₃'s DRBG.

2383 Implementers of an RBGC tree that use siblings for reseeding the DRBG of an RBGC construction
2384 will require a means of recognizing that the parent randomness source is not available and for
2385 the parent's sibling(s) to recognize the validity of the request for the generation of seed material
2386 and the internal state (within the sibling) to be used for the generation process. Additionally,
2387 non-root RBGC constructions cannot guarantee prediction resistance since their randomness
2388 sources cannot provide fresh entropy. However, non-root RBGC constructions **should** be
2389 reseeded periodically to defend against a potential undetected compromise of the internal state.

2390 **7.2. Conceptual Interfaces**

2391 An RBGC construction can support instantiation and generation requests (see Sec. 7.2.1 and
2392 7.2.2, respectively) and may provide a capability to be reseeded (see Sec. 7.2.3).

2393 **7.2.1. RBGC Instantiation**

2394 The DRBG within an RBGC construction may be instantiated by an application at any security
2395 strength possible for the DRBG design that does not exceed the security strength of its
2396 randomness source. This is accomplished using the **DRBG_Instantiate** function discussed in Sec.
2397 2.8.1.1 and SP 800-90A.

2398 The (target) DRBG in an RBGC construction is instantiated by an application using the following
2399 request:

2400 $(status, RBGCx_DRBG_state_handle) =$
2401 $\text{DRBG_Instantiate_request}(s, \text{personalization_string}),$

2402 where s is the requested security strength for the DRBG. The **DRBG_Instantiate_request**
2403 received by the DRBG results in the execution of the **DRBG_Instantiate** function in the DRBG
2404 with the input in the **DRBG_Instantiate_request** provided as input to the **DRBG_Instantiate**
2405 function.

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.

2416 7.2.1.1. Instantiation of the Root RBGC Construction

2417 The randomness source for the root RBGC construction (also referred to as the initial randomness
2418 source) is:

- 2419 • A validated RBG3(XOR) or RBG3(RS) construction, as specified in Sec. 6;

2420 • A validated RBG2(P) or RBG2(NP) construction, as specified in Sec. 5; or

2421 • A validated full-entropy source that is either:

2422 ○ An entropy source that provides output with full entropy, as specified in SP 800-

2423 90B, or

2424 ○ The output of an SP 800-90B-compliant entropy source that has been externally

2425 conditioned by a vetted conditioning function (as specified in Sec. 3.2.2.2) to

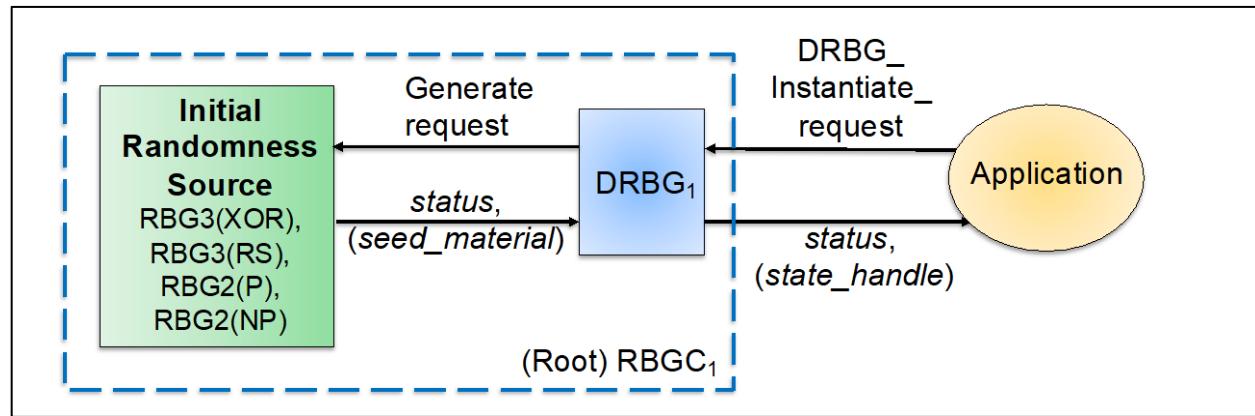
2426 provide output with full entropy.

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.

2436 An RBGC designer must consider how to find an available randomness source and how to access
2437 it.

2438 **7.2.1.1. Instantiating the DRBG in the Root Using an RBG2 or RBG3 Construction as the**
2439 **Initial Randomness Source**



2440

2441 Fig. 30. Instantiation of the DRBG in the root RBGC construction using an RBG2 or RBG3 construction as the
2442 randomness source

2443 Figure 30 depicts a request for instantiation of the root RBGC construction by an application. Let
2444 RBGC₁ be the root and DRBG₁ be its DRBG. In this section, the initial randomness source is either
2445 an RBG2 or RBG3 construction.

2446 Upon receiving a valid instantiation request from an application (see Sec. 7.2.1), the
2447 **DRBG_Instantiate** function within DRBG₁ processes the request by obtaining seed material
2448 from the initial randomness source. Within the **DRBG_Instantiate** function (in DRBG₁), the
2449 randomness source is accessed using a **Get_randomness-source_input** call (see SP 800-90A),
2450 which is replaced as specified below.

2451 Let s be the intended security strength of DRBG₁ in the root RBGC construction.

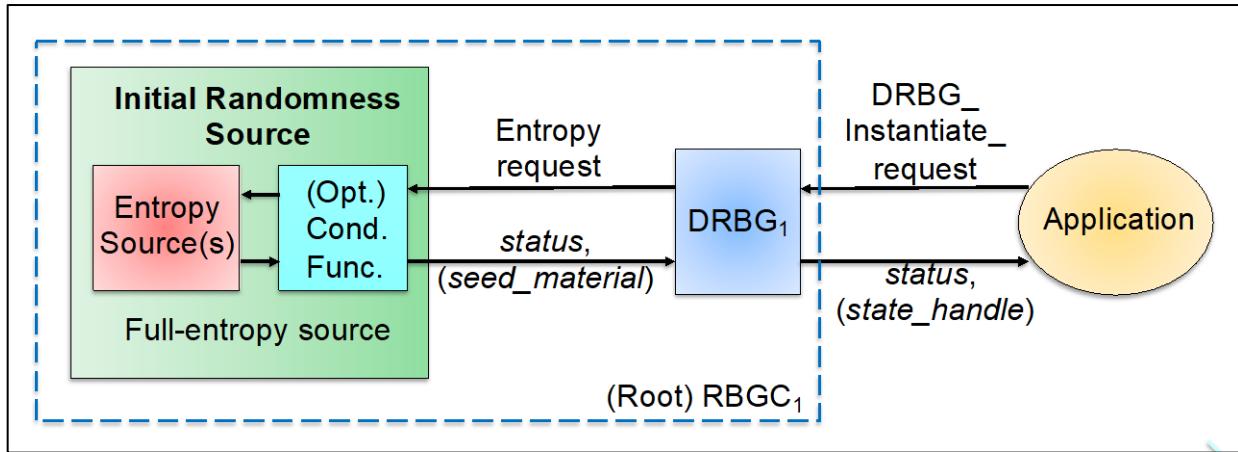
- 2452 1. When the DRBG in the root RBGC construction uses a CTR_DRBG without a derivation
2453 function, $s + 128$ bits²⁴ **shall** be obtained from the initial randomness source.
 - 2454 a. If the randomness source is an RBG2(P) or RBG2(NP) construction, the RBG2
2455 construction **shall** be reseeded before requesting seed material. The
2456 **Get_randomness-source_input** call becomes:
 - 2457 • $status = \text{DRBG_Reseed_request}(RBG2_DRBG_state_handle,$
2458 *additional_input*).
 - 2459 • If ($status \neq \text{SUCCESS}$), 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.

- 2460 • $(status, seed_material) =$
2461 **DRBG_Generate_request**(*RBG2_DRBG_state_handle*,
2462 *s* + 128, *s*, *additional_input*).
- 2463 • If (*status* ≠ SUCCESS), then return (*status, invalid_state_handle*).
- 2464 *RBG2_DRBG_state_handle* is the state handle for the internal state of the DRBG
2465 within the RBG2 construction. Reseed and generate requests received by an RBG2
2466 construction are discussed in Sec. 5.2.3 and 5.2.2, respectively.
- 2467 b. If the randomness source is an RBG3(XOR) or RBG3(RS) construction, the
2468 **Get_randomness-source_input** call becomes:
- 2469 • $(status, seed_material) =$ **RBG3_DRBG_Generate_request**(*s* + 128,
2470 *additional_input*).
- 2471 • If (*status* ≠ SUCCESS), then return (*status, invalid_state_handle*).
- 2472 *RBG3_DRBG_state_handle* is the state handle for the internal state of the DRBG
2473 within the RBG3 construction. An **RBG3_DRBG_Generate_request** received by
2474 an RBG3 construction is discussed in Sec. 6.4.1.2 and 6.5.1.2 (the RBG3(XOR) and
2475 RBG3(RS) constructions, respectively).
- 2476 2. For CTR_DRBG (with a derivation function), Hash_DRBG, and HMAC_DRBG, 3*s*/2 bits
2477 **shall** be obtained from a randomness source that provides a security strength of at least
2478 *s* bits.
- 2479 a. If the randomness source is an RBG2(P) or RBG2(NP) construction, the RBG2
2480 construction **shall** be reseeded before requesting seed material. The
2481 **Get_randomness-source_input** call becomes:
- 2482 • *status* = **DRBG_Reseed**(*RBG2_DRBG_state_handle*, *additional_input*).
- 2483 • If (*status* ≠ SUCCESS), then return (*status, invalid_state_handle*).
- 2484 • $(status, seed_material) =$
2485 **DRBG_Generate_request**(*RBG2_DRBG_state_handle*, 3*s*/2, *s*,
2486 *additional_input*).
- 2487 • If (*status* ≠ SUCCESS), then return (*status, invalid_state_handle*).
- 2488 *RBG2_DRBG_state_handle* is the state handle for the internal state of the DRBG
2489 within the RBG2 construction. Reseed and generate requests received by an RBG2
2490 construction are discussed in Sec. 5.2.3 and 5.2.2, respectively.
- 2491 b. If the randomness source is an RBG3(XOR) or RBG3(RS) construction, the
2492 **Get_randomness-source_input** call becomes:
- 2493 • $(status, seed_material) =$ **RBG3_DRBG_Generate_request**(3*s*/2,
2494 *additional_input*).
- 2495 • If (*status* ≠ SUCCESS), then return (*status, invalid_state_handle*).

2496 *RBG3_DRBG_state_handle* is the state handle for the internal state of the DRBG
2497 within the RBG3 construction. An **RBG3_DRBG_Generate_request** received by
2498 an RBG3 construction is discussed in Sec. 6.4.1.2 and 6.5.1.2 (the RBG3(XOR) and
2499 RBG3(RS) constructions, respectively).

2500 **7.2.1.1.2. Instantiating the Root RBGC Construction Using a Full-Entropy Source as the**
2501 **Randomness Source**

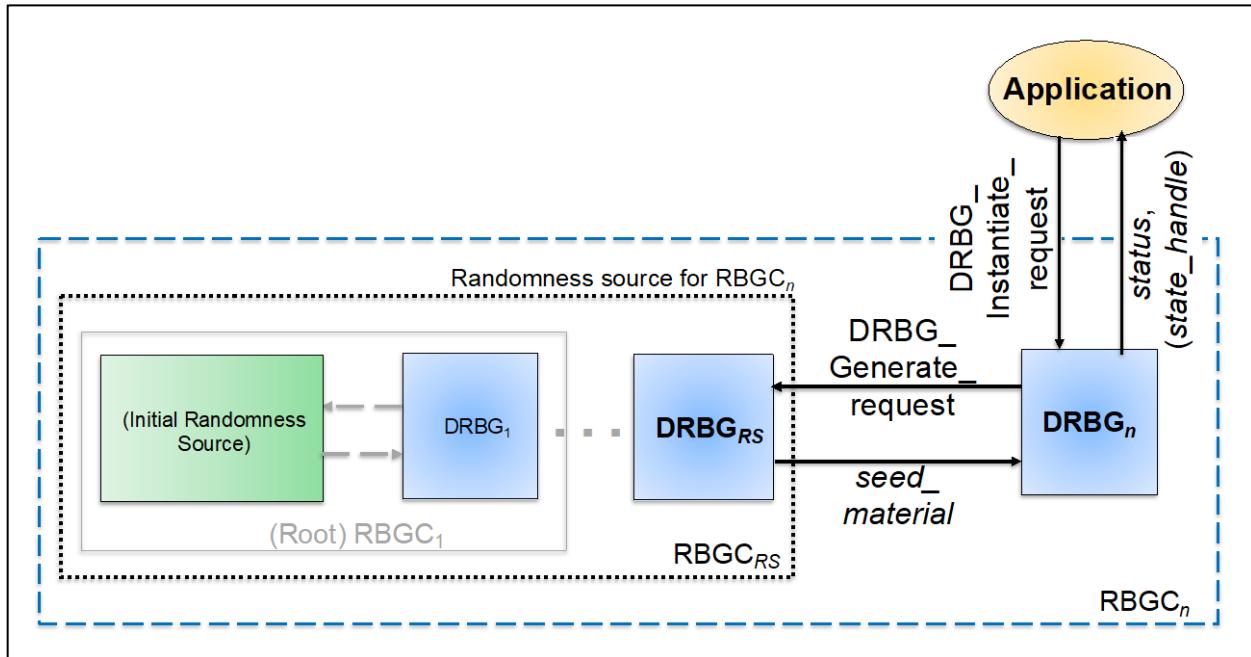


2502
2503 **Fig. 31. Instantiation of the DRBG in the root RBGC construction using a full-entropy source as a randomness**
2504 **source**

2505 Figure 31 depicts a request for instantiation of the root RBGC construction by an application. Let
2506 RBGC₁ be the root and DRBG₁ be its DRBG. In this section, the initial randomness source is a full-
2507 entropy source (see Sec. 7.2.1.1).

2508 Upon receiving a valid instantiation request from an application, the **DRBG_Instantiate** function
2509 within DRBG₁ continues processing the request by obtaining seed material from the full-entropy
2510 source. The full-entropy source may consist of physical or non-physical entropy sources or both,
2511 and either Method 1 or Method 2 may be used to count entropy (see Sec. 2.3). Instantiation is
2512 performed for an RBG2 construction, as specified in Sec. 5.2.1.

2513 **7.2.1.2. Instantiating an RBGC Construction Other Than the Root**



2514

2515 **Fig. 32. Instantiation of the DRBG in RBGC_n using RBGC_{RS} as the randomness source**

2516 Figure 32 depicts a request by an application for the instantiation of the DRBG within an RBGC
2517 construction that is not the root. Let RBGC_n be the RBGC construction receiving the instantiation
2518 request, and let DRBG_n be its DRBG. RBGC_n needs to determine the RBGC construction that will
2519 serve as its randomness source. The randomness source for a DRBG in an RBGC construction that
2520 is not the root of the DRBG chain is the RBGC construction that will immediately precede it in the
2521 chain as its parent. Let RBGC_{RS} be the randomness source for RBGC_n , and let DRBG_{RS} be its DRBG
2522 (see Fig. 32). RBGC_{RS} could be the root RBGC construction. RBGC_1 is outlined in gray in the figure.

2523 Upon receiving a valid instantiation request from an application, such as

2524 $(status, \text{RBGC}_n \text{DRBG}_n \text{state_handle}) =$
2525 **DRBG_Instantiate_request**($s, \text{personalization_string}$),

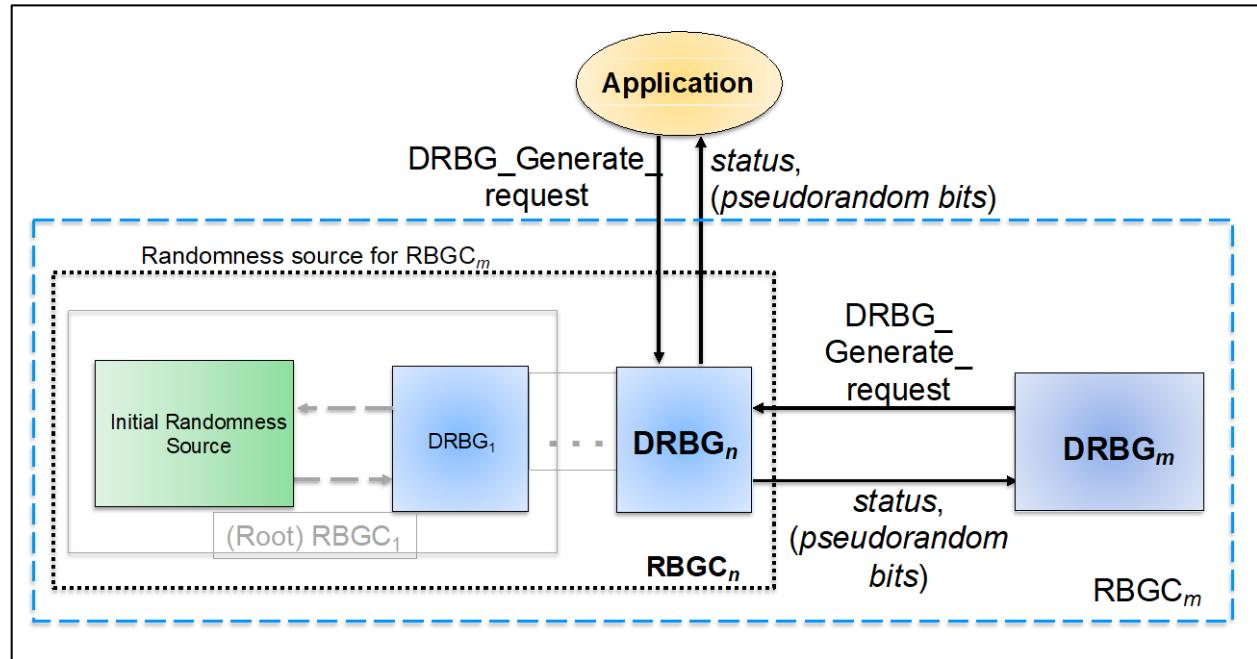
2526 DRBG_n executes its **DRBG_Instantiate** function within DRBG_n and processes the request by
2527 obtaining seed material from its intended parent randomness source (RBGC_{RS}). The
2528 **Get_randomness-source_input** call in the **DRBG_Instantiate** function in DRBG_n is replaced as
2529 specified below.

2530 Let s be the intended security strength of the DRBG in RBGC_n (shown as DRBG_n in the figure).

- 2531 1. When RBGC_n is instantiating a CTR_DRBG without a derivation function, $s + 128$ bits²⁵
2532 shall be obtained from the randomness source (i.e., RBGC_{RS}) by replacing the
2533 **Get_randomness-source_input** call with:
- 2534 • $(status, seed_material) = \text{DRBG_Generate_request}(RBGC_{RS_DRBG_state_handle},$
2535 $s + 128, s, additional_input)$.
- 2536 • If $(status \neq \text{SUCCESS})$, then return $(status, invalid_state_handle)$.
- 2537 $RBGC_{RS_DRBG_state_handle}$ is the state handle for the internal state of the DRBG within
2538 RBGC_{RS} . Upon receiving the **DRBG_Generate_request**, RBGC_{RS} executes its
2539 **DRBG_Generate** function (see Sec. 2.8.1.1 and 7.2.2) and checks its output. That is,
- 2540 • $(status, seed_material) = \text{DRBG_Generate}(RBGC_{RS_DRBG_state_handle},$
2541 $s + 128, s, additional_input)$.
- 2542 • If $(status \neq \text{SUCCESS})$, then return $(status, invalid_state_handle)$.
- 2543 2. For CTR_DRBG (with a derivation function), Hash_DRBG, and HMAC_DRBG, $3s/2$ bits
2544 shall be obtained from the randomness source (RBGC_{RS}) by replacing the
2545 **Get_randomness-source_input** call with:
- 2546 • $(status, seed_material) = \text{DRBG_Generate_request}(RBGC_{RS_DRBG_state_handle},$
2547 $3s/2, s, additional_input)$.
- 2548 • If $(status \neq \text{SUCCESS})$, then return $(status, invalid_state_handle)$.
- 2549 $RBGC_{RS_DRBG_state_handle}$ is the state handle for the internal state of the DRBG within
2550 RBGC_{RS} . Upon receiving the **DRBG_Generate_request**, RBGC_{RS} executes its
2551 **DRBG_Generate** function (see Sec. 2.8.1.1 and 7.2.2) and checks its output. That is,
- 2552 • $(status, seed_material) = \text{DRBG_Generate}(RBGC_{RS_DRBG_state_handle},$
2553 $3s/2, s, additional_input)$.
- 2554 • If $(status \neq \text{SUCCESS})$, then return $(status, invalid_state_handle)$.
- 2555 Section 7.2.2 specifies the behavior of the DRBG in an RBGC construction when it receives a
2556 generate request. The *status* and any generated *seed_material* are returned to the requesting
2557 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.

2558 **7.2.2. Requesting the Generation of Pseudorandom Bits From an RBGC Construction**



2559

2560 **Fig. 33. Generate request received by the DRBG in an RBGC construction**

2561 Figure 33 depicts a generate request received by the DRBG in an RBGC construction (i.e., DRBG_n)
 2562 from a requesting entity (either an application or a DRBG in another RBGC construction,
 2563 shown as DRBG_m and RBGC_m in the figure). When the requesting entity is DRBG_m (rather than an
 2564 application), DRBG_m is attempting to be seeded or reseeded with seed material. DRBG_n **shall** be
 2565 either 1) the parent randomness source for DRBG_m or 2) a sibling of DRBG_m's parent randomness
 2566 source that meets the requirements of an alternative randomness source (see Sec. 7.1.2.2).
 2567 RBGC_n could be the root DRBG (the root is outlined in gray in the figure).

2568 The generate request from the requesting entity for this example is:

2569 $(status, returned_bits) = \text{DRBG_Generate_request}(RBGCn_DRBG_state_handle,$
 2570 $requested_number_of_bits, requested_security_strength, additional_input),$

2571 where *RBGCn_DRBG_state_handle* is the state handle for the internal state of the DRBG in the
 2572 RBGC construction receiving the generate request (RBGC_n). If the **DRBG_Generate_request**
 2573 received by RBGC_n can be handled, the **DRBG_Generate** function in DRBG_n is executed:

2574 $(status, returned_bits) = \text{DRBG_Generate}(RBGCn_DRBG_state_handle,$
 2575 $requested_number_of_bits, requested_security_strength, additional_input).$

2576 The **DRBG_Generate** function within DRBG_n processes the generate request.

- 2577 1. If the generate request cannot be fulfilled (e.g., the requested security strength cannot
 2578 be provided by the DRBG design used in DRBG_n; see SP 800-90A), only an error *status* is
 2579 returned to the requesting entity. No other output is provided.

2580 2. Otherwise, DRBG_n generates the *requested_number_of_bits* and provides them to the
 2581 requesting entity in response to the **DRBG_Generate_request** with a *status* of SUCCESS.

2582 **7.2.3. Reseeding an RBGC Construction**

2583 The reseeding of an RBGC construction is optional. If a reseed capability is implemented within
 2584 the DRBG of an RBGC construction, the RBGC construction may receive a reseed request from an
 2585 application, or the DRBG within the construction may reseed itself based on implementation-
 2586 selected criteria, such as time, number of outputs, events, or — in the case of the root RBGC
 2587 construction using a full-entropy source — the availability of sufficient entropy.

2588 Section 7.2.3.1 discusses the reseeding of the DRBG in the root RBGC construction. Section
 2589 7.2.3.2 discusses the reseeding of the DRBG in an RBGC construction other than the root.

2590 A reseed request from an application is:

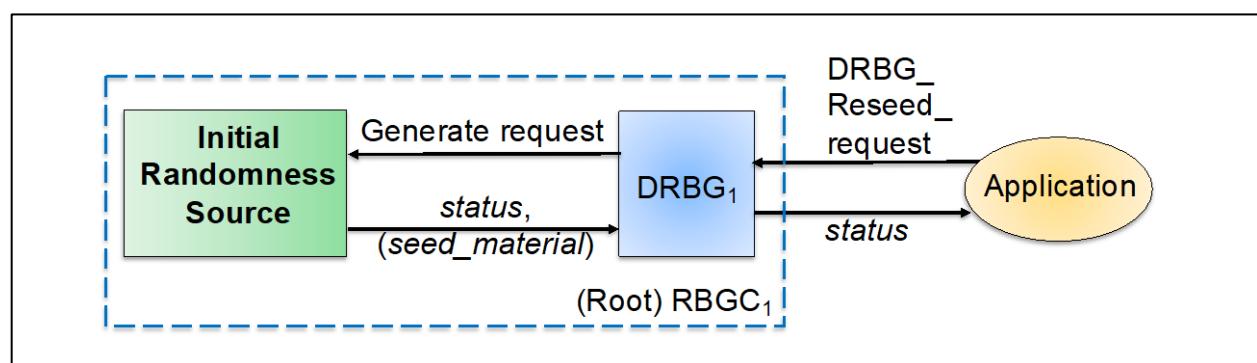
2591 $(status) = \text{DRBG_Reseed_request}(RBGCx_DRBG_state_handle, additional_input)$,

2592 where *RBGCx_DRBG_state_handle* is the state handle for the internal state of the DRBG in the
 2593 RBGC construction receiving the reseed request (*RBGC_x*).²⁶ The **DRBG_Reseed_request** received
 2594 by *RBGC_x* results in the execution of DRBG_x's **DRBG_Reseed** function (see Sec. 2.8.1.3). The
 2595 *status* returned from the **DRBG_Reseed** function **shall** be returned to the application in response
 2596 to the **DRBG_Reseed_request**.

2597 If the reseed request is invalid (e.g., the state handle is not correct or the DRBG does not have a
 2598 reseed capability), an error indication is returned as the *status* to the application (i.e., the DRBG
 2599 has not been reseeded).

2600 Reseeding based on implementation-selected criteria is not initiated by a
 2601 **DRBG_Reseed_request** from an application but is addressed in Sec. 7.2.3.1 and 7.2.3.2.

2602 **7.2.3.1. Reseed of the DRBG in the Root RBGC Construction**

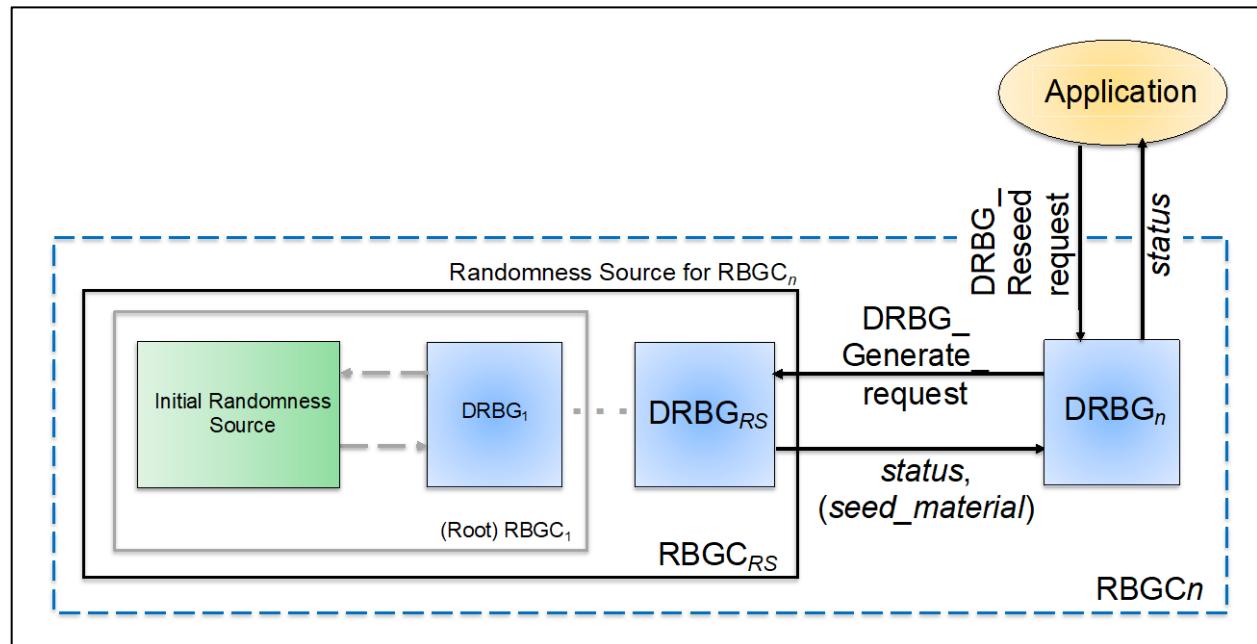


2603
 2604 Fig. 34. Reseed request received by the DRBG in the root RBGC construction

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

- 2605 If the root RBGC construction includes a reseed capability (as shown in Fig. 34), the DRBG in the
2606 root RBGC construction (e.g., RBGC₁) may receive a request from an application for reseeding.
- 2607 Upon the receipt of a valid reseed request or when reseeding is to be performed based on
2608 implementation-selected criteria, the DRBG in the root RBGC construction (e.g., DRBG₁) executes
2609 its **DRBG_Reseed** function to obtain randomness from the initial randomness source for
2610 reseeding itself. This process results in fresh entropy provided by the initial randomness source
2611 so that the next output generated by DRBG₁ has prediction resistance.
- 2612 1. When the DRBG in the root RBGC construction uses the CTR_DRBG without a derivation
2613 function, reseeding is performed in the same manner as for instantiation.
- 2614 • If the initial randomness source is an RBG3(XOR), RBG3(RS), RBG2(P), or RBG2(NP)
2615 construction, input is obtained from the initial randomness source as specified in item
2616 1 of Sec. 7.2.1.1.1.
- 2617 • If the initial randomness source is a full-entropy source, input is obtained as specified
2618 in item 1 of Sec. 7.2.1.1.2.
- 2619 2. When the DRBG in the root RBGC construction uses the CTR_DRBG (with a derivation
2620 function), Hash_DRBG, or HMAC_DRBG, input is obtained from the initial randomness
2621 source in the same manner as for instantiation except that s bits are requested (instead
2622 of $3s/2$ bits), where s is the instantiated security strength of the DRBG in the root.
- 2623 • If the initial randomness source is an RBG3(XOR), RBG3(RS), RBG2(P), or RBG2(NP)
2624 construction, input is obtained from the initial randomness source as specified in item
2625 2 of Sec. 7.2.1.1.1.
- 2626 • If the initial randomness source is full-entropy source, input is obtained as specified
2627 in item 2 of Sec. 7.2.1.1.2.

2628 **7.2.3.2. Reseed of the DRBG in an RBGC Construction Other Than the Root**



2629

2630 **Fig. 35. Reseed request received by an RBGC construction other than the root**

2631 As shown in Fig. 35, a DRBG in an RBGC construction other than the root (e.g., RBGC_n) may receive
2632 a request for reseeding from an application. DRBG_n may also reseed itself based on
2633 implementation-selected criteria.

2634 Let DRBG_{RS} be the randomness source to be used for reseeding. DRBG_{RS} **must** be either DRBG_n's
2635 parent randomness source or a sibling of the parent (see Sec. 7.1.2.2). DRBG_{RS} may be the DRBG
2636 of the root RBGC construction (outlined in gray in the figure). Prediction resistance is not
2637 provided for the DRBG being reseeded (DRBG_n) since fresh entropy is not provided by the
2638 randomness source in this case (DRBG_{RS}).

2639 Upon the receipt of a valid reseed request or when a reseed is to be performed based on
2640 implementation-selected criteria, the DRBG in RBGC_n executes its **DRBG_Reseed** function (if
2641 implemented). The **Get_randomness-source_input** request in the **DRBG_Reseed** function is
2642 replaced by the following:

- 2643 • $(status, seed_material) = \text{DRBG_Generate_request}(RBGC_{RS_DRBG_state_handle}, s, s, additional_input)$.
- 2645 • If $(status \neq \text{SUCCESS})$, then return $(status, invalid_bitstring)$,

2646 where $RBGC_{RS_DRBG_state_handle}$ is the state handle for the internal state of the DRBG in the
2647 randomness source (i.e., RBGC_{RS}). Upon receiving the request, RBGC_{RS} executes its
2648 **DRBG_Generate** function. A *status* indication will be returned from RBGC_{RS} along with seed
2649 material if the *status* indicates a success (see Sec. 7.2.2).

2650 Upon the receipt of a response from the randomness source (RBG_{RS}), the DRBG in RBGC_n
2651 proceeds as follows:

- 2652 1. If an error indicator is received from the randomness source (RBG_{RS}) in response to the
2653 generate request, the error indicator is forwarded to the application as the *status* in the
2654 response to the reseed request.
- 2655 2. If an error indicator is not received from the randomness source (i.e., RBG_{RS}) and
2656 *seed_material* is provided, the *seed_material* is incorporated into the internal state of the
2657 DRBG in RBGC_n as specified in its **DRBG_Reseed** function. If the reseeding of the DRBG
2658 in RBGC_n was in response to a **DRBG_Reseed_request** from an application, the *status*
2659 received from the randomness source is returned to the application.

2660 **7.3. RBGC Requirements**

2661 **7.3.1. General RBGC Construction Requirements**

2662 An RBGC construction has the following general testable requirements (i.e., testable by the
2663 validation labs):

- 2664 1. An **approved** DRBG from SP 800-90A whose components are capable of providing the
2665 targeted security strength for an RBGC construction **shall** be employed.
- 2666 2. RBGC components **shall** be successfully validated for compliance with SP 800-90A, SP 800-
2667 90B, SP 800-90C, FIPS 140, and the specification of any other **approved** algorithm used
2668 within the RBGC construction, as applicable.
- 2669 3. An RBGC construction **shall not** produce any output until it is instantiated.
- 2670 4. An RBGC construction **shall not** provide output for generating requests that specify a
2671 security strength greater than the instantiated security strength of its DRBG.
- 2672 5. If a health test on the DRBG in an RBGC construction fails, the DRBG instantiation **shall** be
2673 terminated.
- 2674 6. The seed material provided to the DRBG within an RBGC construction **shall** remain secret
2675 during transfer from the DRBG's randomness source and remain unobservable from
2676 outside its RBG boundary.
- 2677 7. The internal state of the DRBG within an RBGC construction **shall** remain unobservable
2678 from outside its RBG boundary.
- 2679 8. A tree of RBGC constructions and the initial randomness source for the root RBGC
2680 construction **shall** be implemented and operated on a single, physical platform. See
2681 Appendix A.3 for further discussion.
- 2682 9. The initial randomness source **shall not** be removable from the computing platform
2683 during operation. If a replacement is required, the root **shall** be instantiated using the
2684 replaced randomness source.

2685 10. The seed material **shall not** be output from the computing platform on which it was
2686 generated.

2687 11. The internal state of the DRBG within an RBGC construction **shall not** be removed from
2688 the computing platform on which it was created, including for storage, and **shall** only be
2689 available to the DRBG instantiation for which it was created.

2690 12. If the (parent) randomness source for an RBGC construction is not available for reseeding,
2691 the DRBG in the RBGC construction may continue to generate output without reseeding
2692 or may be reseeded using a sibling of the parent that has been appropriately validated.
2693 When used as an alternative randomness source for reseeding, the sibling **shall** have been
2694 validated as an RBGC construction.

2695 General requirements for an RBGC construction that are non-testable are:

2696 13. Each RBGC construction **must** be able to determine the type of randomness source
2697 available for its use and how to access it.

2698 14. The randomness source for an RBGC construction **must** provide the requested number of
2699 bits at a security strength of s bits or higher, where s is the targeted security strength for
2700 that RBGC construction.

2701 15. The specific output of the randomness source (or portion thereof) that is used for the
2702 instantiation or reseed of an RBGC construction **must not** be used for any other purpose,
2703 including for seeding or reseeding a different instantiation or RBGC construction.

2704 16. The output of an RBGC construction **must not** be used as seed material for a predecessor
2705 (e.g., ancestor) RBGC construction.

2706 7.3.2. Additional Requirements for the Root RBGC Construction

2707 An RBGC construction that is used as the root of a DRBG chain has the following additional
2708 testable requirements (i.e., testable by the validation labs):

- 2709 1. For CTR_DRBG (with a derivation function), Hash_DRBG, or HMAC_DRBG, $3s/2$ bits
2710 **shall** be obtained from the initial randomness source for instantiation, where s is the
2711 targeted security strength for the DRBG used in the RBGC construction. When reseeding,
2712 s bits **shall** be obtained from the initial randomness source.
- 2713 2. For a CTR_DRBG without a derivation function used as the DRBG within the root RBGC
2714 construction, $s + 128$ bits²⁷ **shall** be obtained from the randomness source for
2715 instantiation and reseeding, where s is the targeted security strength for the DRBG used
2716 in the RBGC construction.

²⁷ Note that $s + 128 = \text{keylen} + \text{blocklen} = \text{seedlen}$, as specified in SP 800-90Ar1.

2717 3. If the randomness source for the root RBGC construction is an RBG2 construction, a
2718 request for reseeding the DRBG in the RBG2 construction **shall** precede a request for
2719 generating seed material.

2720 The non-testable requirements for the root RBGC construction are:

- 2721 4. The initial randomness source for the root RBGC construction **must** be a validated
2722 RBG3(XOR), RBG3(RS), RBG2(P), or RBG2(NP) construction or a full-entropy source.
2723 5. A full-entropy source serving as the initial randomness source **must** be either an entropy
2724 source that has been validated as providing full-entropy output or a validated entropy
2725 source that uses the external conditioning function specified in Sec. 3.2.2.2.
2726 6. The DRBG in the root RBGC construction may be instantiated at any security strength for
2727 the design, subject to the following restriction: if the initial randomness source is an
2728 RBG2(P) or RBG2(NP) construction, the root **must not** be instantiated at a security
2729 strength greater than the security strength of the RBG2(P) or RBG2(NP) construction.

2730 **7.3.3. Additional Requirements for an RBGC Construction That is NOT the Root of a DRBG Chain**

2731 An RBGC construction that is NOT the root of a DRBG chain has no additional testable
2732 requirements beyond those in Sec. 7.3.1.

2733 The non-testable requirements for an RBGC construction that is not the root of a DRBG chain are:

- 2734 1. Each RBGC construction **must** have only one parent RBGC construction as a randomness
2735 source for instantiation and reseeding, although under certain conditions, a sibling of the
2736 parent may be used as a randomness source for reseeding (see requirement 12 in Sec.
2737 7.3.1).
2738 2. An RBGC construction **must** reside on the same computing platform as its parent and any
2739 alternative randomness source.
2740 3. Each RBGC construction may be instantiated at any security strength for the design that
2741 does not exceed the security strength of its parent randomness source.

2743 **8. Testing**

2744 Two types of testing are specified in this recommendation: health testing and implementation-
2745 validation testing. Health testing **shall** be performed on all RBGs that claim compliance with this
2746 recommendation (see Sec. 8.1). Section 8.2 provides requirements for implementation
2747 validation.

2748 **8.1. Health Testing**

2749 *Health testing* is the testing of an implementation prior to and during normal operations to
2750 determine whether the implementation continues to perform as expected and as validated.
2751 Health testing is performed by the RBG itself (i.e., the tests are designed into the RBG
2752 implementation).

2753 An RBG **shall** support the health tests specified in SP 800-90A and SP 800-90B as well as perform
2754 health tests on the components of SP 800-90C. FIPS 140 specifies the testing to be performed
2755 within a cryptographic module.

2756 **8.1.1. Testing RBG Components**

2757 Whenever an RBG receives a request to start up or perform health testing, a request for health
2758 testing **shall** be issued to the RBG components (e.g., the DRBG and any entropy source).

2759 **8.1.2. Handling Failures**

2760 Failures may occur during the use of entropy sources and during the operation of other
2761 components of an RBG.

2762 SP 800-90A and SP 800-90B discuss error handling for DRBGs and entropy sources, respectively.

2763 **8.1.2.1. Entropy-Source Failures**

2764 A failure of a validated entropy source is reported to the **Get_entropy_bitstring** process in
2765 response to entropy requests to the entropy source(s). The **Get_entropy_bitstring** function
2766 notifies the consuming application of such failures as soon as possible (see item 4 of Sec. 3.1).
2767 The consuming application may choose to terminate the RBG operation. Otherwise, the RBG may
2768 continue operation if any entropy source credited for providing entropy²⁸ is still healthy (i.e., a
2769 failure has not been reported by those entropy sources).

2770 If all entropy sources credited with providing entropy report failures, the RBG operation **shall** be
2771 terminated (e.g., stopped) until such time as the entropy source is repaired and successfully
2772 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.

2773 **8.1.2.2. Failures by Non-Entropy-Source Components**

2774 Failures by non-entropy-source components may be caused by either hardware or software
2775 failures. Some of these may be detected using known-answer health tests within the RBG.
2776 Failures could also be detected by the system in or on which the RBG resides.

2777 When such failures are detected that affect the RBG, the RBG operation **shall** be terminated. The
2778 RBG **must not** resume operations until the reasons for the failure have been determined, the
2779 failure has been repaired, and the RBG successfully tested for proper operation.

2780 **8.2. Implementation Validation**

2781 Implementation validation is the process of verifying that an RBG and its components fulfill the
2782 requirements of this recommendation. Validation is accomplished by:

- 2783 • Validating the components from SP 800-90A and SP 800-90B
- 2784 • Validating the use of the constructions in SP 800-90C via code inspection, known answer
2785 tests, or both, as appropriate
- 2786 • Validating that the appropriate documentation has been provided, as specified in SP 800-
2787 90C

2788 Documentation **shall** be developed that will provide assurance to testers that an RBG that claims
2789 compliance with this recommendation has been implemented correctly. This documentation
2790 **shall** include the following as a minimum:

- 2791 • An identification of the constructions and components used by the RBG, including a
2792 diagram of the interaction between the constructions and components.
- 2793 • If an external conditioning function is used, an indication of the type of conditioning
2794 function and the method for obtaining any keys that are required by that function.
- 2795 • Appropriate documentation, as specified in SP 800-90A and SP 800-90B. The DRBG and
2796 the entropy sources **shall** be validated for compliance with SP 800-90A or SP 800-90B,
2797 respectively, and the validations successfully finalized before the completion of RBG
2798 implementation validation.
- 2799 • The maximum security-strength that can be supported by the DRBG.
- 2800 • A description of all validated and non-validated entropy sources used by the RBG,
2801 including identifying whether the entropy source is a physical or non-physical entropy
2802 source.
- 2803 • Documentation justifying the independence of all validated entropy sources from all
2804 other validated and non-validated entropy sources employed.
- 2805 • An identification of the features supported by the RBG (e.g., access to the underlying
2806 DRBG of an RBG3 construction).

- 2807 • A description of the health tests performed, including an identification of the periodic
2808 intervals for performing the tests.
- 2809 • A description of any support functions other than health testing.
- 2810 • A description of the RBG components within the RBG security boundary (see Sec. 2.5).
- 2811 • For an RBG1 construction, a statement indicating that the randomness source **must** be a
2812 validated RBG2(P) or RBG3 construction (e.g., this could be provided in user
2813 documentation and/or in a security policy).
- 2814 • If sub-DRBGs can be used in an RBG1 construction, the maximum number of sub-DRBGs
2815 that can be supported by the implementation and the security strengths to be supported
2816 by the sub-DRBGs.
- 2817 • For RBG2 and RBG3 constructions, a statement that identifies the conditions under which
2818 the DRBG is reseeded (e.g., when requested by a consuming application, at a given time
2819 interval, etc.).
- 2820 • For an RBG3 construction, a statement that indicates whether the DRBG can be accessed
2821 directly (i.e., the DRBG internal state used by the RBG3 construction can be accessed using
2822 calls directly to the DRBG).
- 2823 • For an RBG3 construction, the security policy **shall** indicate the fallback security strength
2824 that can be supported by the DRBG if the entropy source fails (i.e., the fallback security
2825 strength is the instantiated security strength of the DRBG).
- 2826 • For an RBG3(RS) construction, when implementing CTR_DRBG (with a derivation
2827 function), Hash_DRBG, or HMAC_DRBG, the method used for obtaining $s + 64$ bits of
2828 entropy to produce s full-entropy bits (see Sec. 6.5.1.2.1)
- 2829 • For an RBGC construction, whether it is capable of serving as the root of a DRBG chain,
2830 how it “finds” an appropriate randomness source for seeding and reseeding (if
2831 implemented), whether it can instantiate child RBGC constructions, any restrictions on
2832 the number of child RBGC constructions in the implementation, whether it can be used
2833 as an alternative randomness source for another RBGC construction and how this is
2834 accomplished (see the note in Sec. 7.1.2.2), and whether it can be reseeded.
- 2835 • If an RBGC construction can serve as the root of a DRBG chain, identify the initial
2836 randomness source types that can be used. If the randomness source can be a full-entropy
2837 source, describe the entropy sources to be used.
- 2838 • Documentation specifying the guidance to users about fulfilling the non-testable
2839 requirements, as appropriate (see Sec. 4.4, 5.3, 6.3, and 7.3).

2840 **References**

- 2841 [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
- 2842 [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>.
- 2843 [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>.
- 2844 [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>.
- 2845 [FIPS_140] National Institute of Standards and Technology (2001) *Security Requirements for Cryptographic Modules*. (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 140-2, Change Notice 2 December 03, 2002. <https://doi.org/10.6028/NIST.FIPS.140-2>
- 2846 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>
- 2847 [FIPS_140IG] National Institute of Standards and Technology, Canadian Centre for Cyber Security *Implementation Guidance for FIPS 140-2 and the Cryptographic Module Validation Program*, [Amended]. Available at <https://csrc.nist.gov/csrc/media/projects/cryptographic-module-validation-program/documents/fips140-2/FIPS1402IG.pdf>
- 2848 [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>
- 2849 [FIPS_197] National Institute of Standards and Technology (2001) *Advanced Encryption Standard (AES)*. (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 197. <https://doi.org/10.6028/NIST.FIPS.197>
- 2850 [FIPS_198] National Institute of Standards and Technology (2008) *The Keyed-Hash Message Authentication Code (HMAC)*. (U.S. Department of Commerce,

- 2881 Washington, DC), Federal Information Processing Standards Publication
2882 (FIPS) 198-1. <https://doi.org/10.6028/NIST.FIPS.198-1>.
- 2883 [FIPS_202] National Institute of Standards and Technology (2015) *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. (U.S. Department of Commerce, Washington, DC), Federal Information Processing Standards Publication (FIPS) 202. <https://doi.org/10.6028/NIST.FIPS.202>
- 2888 [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>
- 2891 [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>.
- 2895 [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>
- 2899 [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>
- 2904 [SP_800-38B] Dworkin MJ (2005) *Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authentication*. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-38B, Includes updates as of October 6, 2016. <https://doi.org/10.6028/NIST.SP.800-38B>
- 2909 [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>
- 2913 [SP_800-90A] Barker EB, Kelsey JM (2015) *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. (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>
- 2917 [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 Generation*. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-90B. <https://doi.org/10.6028/NIST.SP.800-90B>
- 2922 [SP_800-131A] Barker EB, Roginsky AL (2019) *Transitioning the Use of Cryptographic Algorithms and Key Lengths*. (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

2931 **Appendix A. Auxiliary Discussions (Informative)**

2932 **A.1. Entropy vs. Security Strength**

2933 This appendix compares and contrasts the concepts of *entropy* and *security strength*.

2934 **A.1.1. Entropy**

2935 Suppose that an entropy source produces n -bit strings with m bits of entropy in each bitstring.
2936 This means that when an n -bit string is obtained from that entropy source, the best possible
2937 guess of the value of the string has a probability of no more than 2^{-m} of being correct.

2938 Entropy can be thought of as a property of a probability distribution, like the mean or variance.
2939 Entropy measures the unpredictability or randomness of the *probability distribution on bitstrings*
2940 *produced by the entropy source*, not a property of any particular bitstring. However, the
2941 terminology is sometimes slightly abused by referring to a bitstring as having m bits of entropy.
2942 This simply means that the bitstring came from a source that ensures m bits of entropy in its
2943 output bitstrings.

2944 Because of the inherent variability in the process, predicting future entropy-source outputs does
2945 not depend on an adversary's amount of computing power.

2946 **A.1.2. Security Strength**

2947 A deterministic cryptographic mechanism (e.g., the DRBGs defined in SP 800-90A) has a security
2948 strength — a measure of how much computing power an adversary expects to need to defeat
2949 the security of the mechanism. If a DRBG has an s -bit security strength, an adversary who can
2950 make 2^w computations of the underlying block cipher or hash function, where $w < s$, expects to
2951 have about a 2^{w-s} probability of defeating the DRBG's security. For example, an adversary who
2952 can perform 2^{96} AES encryptions can expect to defeat the security of the CTR-DRBG that uses
2953 AES-128 with a probability of about 2^{-32} (i.e., 2^{96-128}).

2954 **A.1.3. A Side-by-Side Comparison**

2955 Informally, one way of thinking of the difference between security strength and entropy is the
2956 following: suppose that an adversary somehow obtains the internal state of an entropy source
2957 (e.g., the state of all the ring oscillators and any internal buffer). This might allow the adversary
2958 to predict the next few bits from the entropy source (assuming that there is some buffering of
2959 bits within the entropy source), but the entropy source outputs will once more become
2960 unpredictable to the adversary very quickly. For example, knowing what faces of the dice are
2961 currently showing does not allow a player to successfully predict the next roll of the dice.

2962 In contrast, suppose that an adversary somehow obtains the internal state of a DRBG. Because
2963 the DRBG is deterministic, the adversary can then predict all future outputs from the DRBG until
2964 the next reseeding of the DRBG with a sufficient amount of entropy.

2965 An entropy source provides bitstrings that are hard for an adversary to guess correctly but usually
2966 have some detectable statistical flaws (e.g., they may have slightly biased bits, or successive bits
2967 may be correlated). However, a well-designed DRBG provides bitstrings that exhibit none of these
2968 properties. Rather, they have independent and identically distributed bits, with each bit taking
2969 on a value with a probability of exactly 0.5. These bitstrings are only unpredictable to an
2970 adversary who does not know the DRBG’s internal state and is computationally bounded.

2971 **A.1.4. Entropy and Security Strength in This Recommendation**

2972 The DRBG within the RBG1 construction is instantiated from either an RBG2(P) or an RBG3
2973 construction. To instantiate the RBG1 construction at a security strength of s bits, this
2974 recommendation requires the source RBG to support a security strength of at least s bits and
2975 provide a bitstring that is $3s/2$ bits long for most of the DRBGs. However, for a CTR_DRBG
2976 without a derivation function, a bitstring that is $s + 128$ bits long is required. An RBG3
2977 construction supports any desired security strength.

2978 The DRBG within an RBG2 or RBG3 construction is instantiated using a bitstring with a certain
2979 amount of entropy obtained from a validated entropy source.²⁹ In order to instantiate the DRBG
2980 to support an s -bit security strength, a bitstring with at least $3s/2$ bits of entropy is required for
2981 the instantiation of most of the DRBGs. Reseeding requires a bitstring with at least s bits of
2982 entropy. However, instantiating and reseeding a CTR_DRBG without a derivation function
2983 requires a bitstring with exactly $s + 128$ full-entropy bits. This bitstring can either be obtained
2984 directly from an entropy source that provides full-entropy output or from an entropy source via
2985 an **approved** (i.e., vetted) conditioning function (see Sec. 3.2).

2986 RBG3 constructions are designed to provide full-entropy outputs but with a DRBG included in the
2987 design as a second security anchor in case the entropy source fails undetectably. Entropy bits are
2988 obtained either directly from an entropy source or from an entropy source via an **approved** (i.e.,
2989 vetted) conditioning function. When the entropy source is working properly, an n -bit output from
2990 the RBG3 construction is said to provide n bits of entropy. The DRBG in an RBG3 construction is
2991 always required to support the highest security strength that can be provided by its design
2992 (*highest_strength*). If an entropy-source has an undetectable failure, the RBG3 construction
2993 outputs are generated at that security strength. In this case, the security strength of a bitstring — that
2994 is, *security_strength* = **min**(*highest_strength*, *length*).

2996 The DRBG within an RBGC construction is instantiated using a bitstring from a randomness
2997 source. The randomness source for an RBGC construction will be either an initial randomness
2998 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.

2999 construction. The tree of RBGC constructions will always originate from an **approved** initial
 3000 randomness source that is either a full-entropy source or an RBG2 or RBG3 construction, each of
 3001 which includes a validated entropy source.

3002 In conclusion, entropy sources and properly functioning RBG3 constructions provide output with
 3003 entropy. RBG1, RBG2, and RBGC constructions provide output with a security strength that
 3004 depends on the security strength of the RBG instantiation and the length of the output. Likewise,
 3005 if the entropy source used by an RBG3 construction fails undetectably, the output is then
 3006 dependent on the DRBG within the construction (i.e., an RBG(P) construction) to produce output
 3007 at the highest security strength for the DRBG design.

3008 Because of the difference between the use of “entropy” to describe the output of an entropy
 3009 source and the use of “security strength” to describe the output of a DRBG, the term
 3010 “randomness” is used as a general term to mean either “entropy” or “security strength,” as
 3011 appropriate. A “randomness source” is the general term for an entropy source or RBG that
 3012 provides the randomness used by an RBG.

3013 **A.2. Generating Full-Entropy Output Using the RBG3(RS) Construction**

3014 Table 4 provides information on generating full-entropy output using the RBG3(RS) construction
 3015 with the DRBGs in SP 800-90A.

3016 **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

3017 Each DRBG is based on the use of an **approved** hash function or block cipher algorithm as a
 3018 cryptographic primitive.

- 3019 • Column 1 lists the DRBG types.
 3020 • Column 2 identifies the cryptographic primitives that can be used by the DRBG(s) in
 3021 column 1.

- 3022 • Column 3 indicates the highest security strength (s) that can be supported by the
3023 cryptographic primitive in column 2.³⁰
- 3024 • Column 4 indicates the amount of fresh entropy (r) that is obtained by a **DRBG_Reseed**
3025 function for the security strength identified in column 3, as specified in SP 800-90A.
- 3026 • Column 5 indicates the amount of entropy required to be inserted into the cryptographic
3027 primitive ($s + 64$) to produce s bits with full entropy.
- 3028 For the CTR_DRBG with no derivation function, the amount of entropy obtained during a
3029 reseed as specified in SP 800-90A (see column 4) exceeds the amount of entropy needed to
3030 subsequently generate s bits of output with full entropy (see column 5), where s is 128, 192, or
3031 256. Therefore, reseeding as specified in SP 800-90A is appropriate.
- 3032 However, for the CTR_DRBG that uses a derivation function or the Hash_DRBG or
3033 HMAC_DRBG, a reseed as specified in SP 800-90A does not provide sufficient entropy for
3034 producing s bits of full-entropy output for each execution of the **DRBG_Generate** function (see
3035 columns 4 and 5). Section 6.5.1.2.1 provides two methods for obtaining the required $s + 64$ bits
3036 of entropy needed to generate s bits of full-entropy output:
- 3037 1. Modify the **DRBG_Reseed** function to obtain $s + 64$ bits of entropy from the entropy
3038 source(s) rather than the s bits of entropy specified in SP 800-90A. This approach may be
3039 used in implementations that have access to the internals of the DRBG implementation.
- 3040 2. Obtain 64 bits of entropy directly from the entropy source(s) and provide it as additional
3041 input when invoking the **DRBG_Reseed** function. As specified in SP 800-90A, the
3042 **DRBG_Reseed** function obtains s bits of entropy from the entropy source(s) and
3043 concatenates the additional input to it before updating the internal state with the
3044 concatenated result (see the specification for the reseed algorithm for each DRBG type in
3045 SP 800-90A), thus incorporating $s + 64$ bits of fresh entropy into the DRBG's internal state.

3046 A.3. Additional Considerations for RBGC Constructions

- 3047 The boundaries for an RBGC construction are more difficult to define than other constructions
3048 specified in this document, which makes validation more difficult. This difficulty arises from
3049 changes in the structure of the RBGC tree (e.g., RBGC constructions created in software at
3050 runtime) and the possibility that the module containing the DRBG of the RBGC construction may
3051 be validated separately from the module containing the randomness source that seeds and
3052 reseeds it.
- 3053 This section contains examples of acceptable RBGC constructions as well as designs that properly
3054 transmit seed material. To simplify the discussion, the figures show only the DRBG in each RBGC
3055 construction. For example, DRBG₁ is the DRBG for the RBGC₁, which is used in the examples as
3056 the root of the tree (i.e., the root DRBG), and DRBG₂ is the DRBG for RBGC₂.

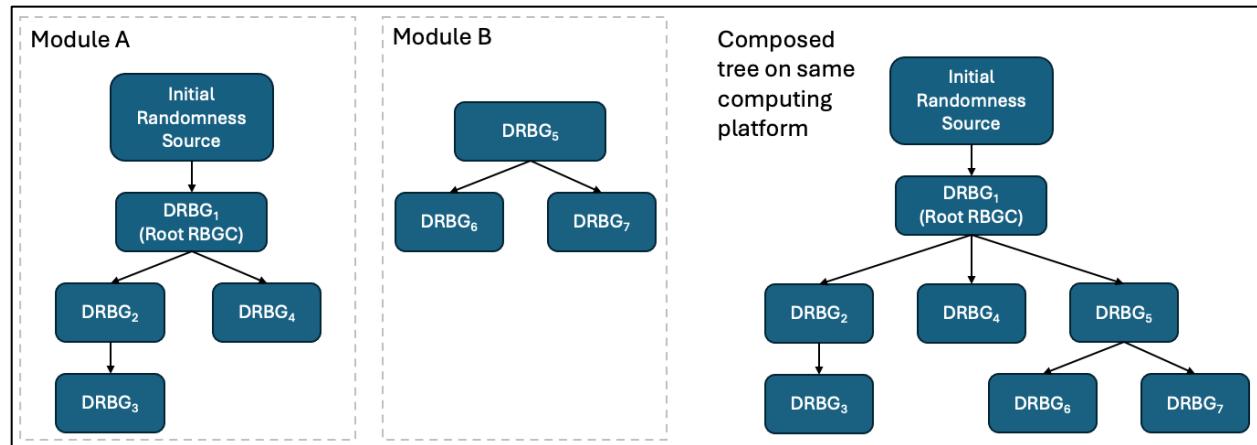
³⁰ Columns 2 and 3 provide the same information as **Table 3**.

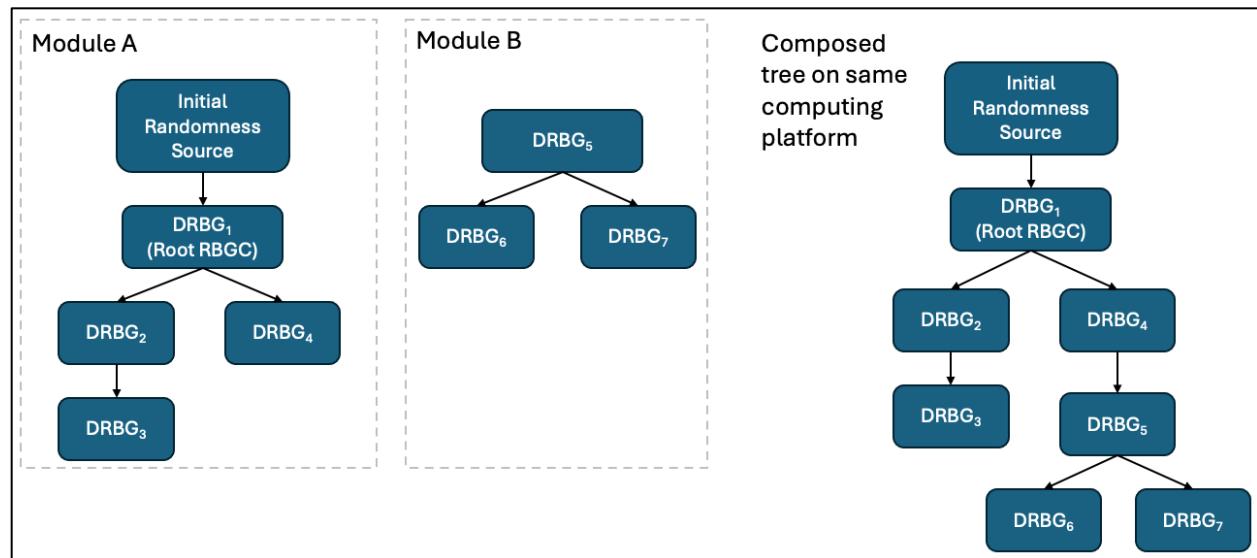
3057 **A.3.1. RBGC Tree Composition**

3058 When parts of an RBGC tree are validated separately, the tree can later be composed in a safe
 3059 manner to ensure that the requirements given in Sec. 7 are met. An RBGC tree consists of an
 3060 initial randomness source and a root RBGC construction (at a minimum) and may include
 3061 descendent RBGC constructions (e.g., children and grandchildren). Additional RBGC
 3062 constructions (called subtrees) may be added to form a more complex tree. Each subtree consists
 3063 of at least one RBGC construction that may have its own descendants but is unable to access the
 3064 initial randomness source.

3065 Consider two modules — A and B — that are evaluated separately (see Fig. 36). Module B does
 3066 not contain a root DRBG, but module A does. Module A contains an initial randomness source
 3067 and a DRBG that can access the initial randomness source to serve as the root of a tree (shown
 3068 as DRBG₁). Module B does not include an initial randomness source, so no DRBG in that module
 3069 can serve as a root. The following examples show how DRBGs in module B can be evaluated as
 3070 RBGC constructions.

3071 The simplest case for tree composition occurs when one RBGC construction satisfies the
 3072 requirements for the root RBGC, and every other RBGC construction involved meets the
 3073 requirements of a non-root RBGC construction. Figures 36 and 37 show compositions where
 3074 module A has been validated as an RBGC tree containing an initial randomness source, a root
 3075 (shown as DRBG₁), two children of the root (DRBG₂ and DRBG₄), and DRBG₃ (a child of DRBG₂).
 3076 Module B contains a subtree consisting of DRBG₅ and two child DRBGs (DRBG₆ and DRBG₇). In
 3077 these examples, all DRBGs meet the requirements for RBGC constructions.

3078
3079 **Fig. 36. Subtree in module B seeded by root RBGC of module A**

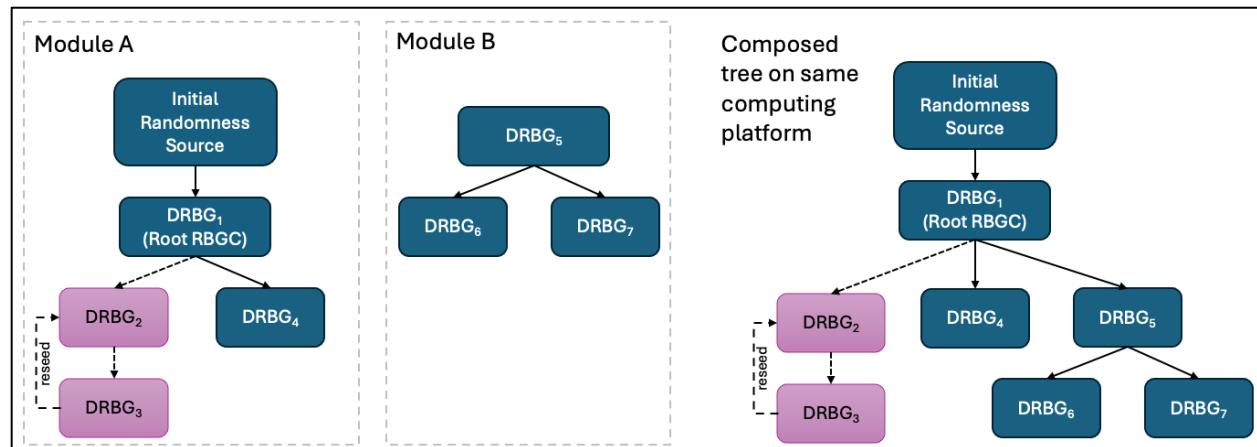


3080

3081 **Fig. 37. Subtree in module B seeded by a non-root DRBG of module A (i.e., DRBG₄)**

3082 In Fig. 36, the DRBGs in module B are added to the tree by using the root (**DRBG₁**) as the
 3083 randomness source for **DRBG₅**. In Fig. 37, the DRBGs in module B are added to the tree by using
 3084 **DRBG₄** as the randomness source for **DRBG₅**.

3085 It is possible to compose trees where some of the DRBGs in module A do not meet the
 3086 requirements of an RBGC-compliant tree. Figure 38 depicts two DRBGs — **DRBG₂** and **DRBG₃** —
 3087 that do not meet RBGC requirements because a loop exists when **DRBG₃** is used to reseed **DRBG₂**.
 3088 The DRBGs in purple boxes connected to the parent through dashed lines do not meet the DRBG
 3089 requirements for an RBGC construction.



3090

3091 **Fig. 38. Subtree in module B seeded by DRBG₄ in module A**

3092 If module B is added to the tree such that **DRBG₄** is the randomness source for **DRBG₅**, the
 3093 elements of module B's subtree only depend on DRBGs that meet RBGC requirements (i.e.,
 3094 **DRBG₁** and **DRBG₄**) and may therefore be validated as RBGC constructions when added to the
 3095 tree in this manner.

3096 However, if the DRBGs in module B are added to the tree so that DRBG₂ is the randomness source
3097 for DRBG₅ (see Fig. 39), then the resulting tree is not a compliant RBGC tree.

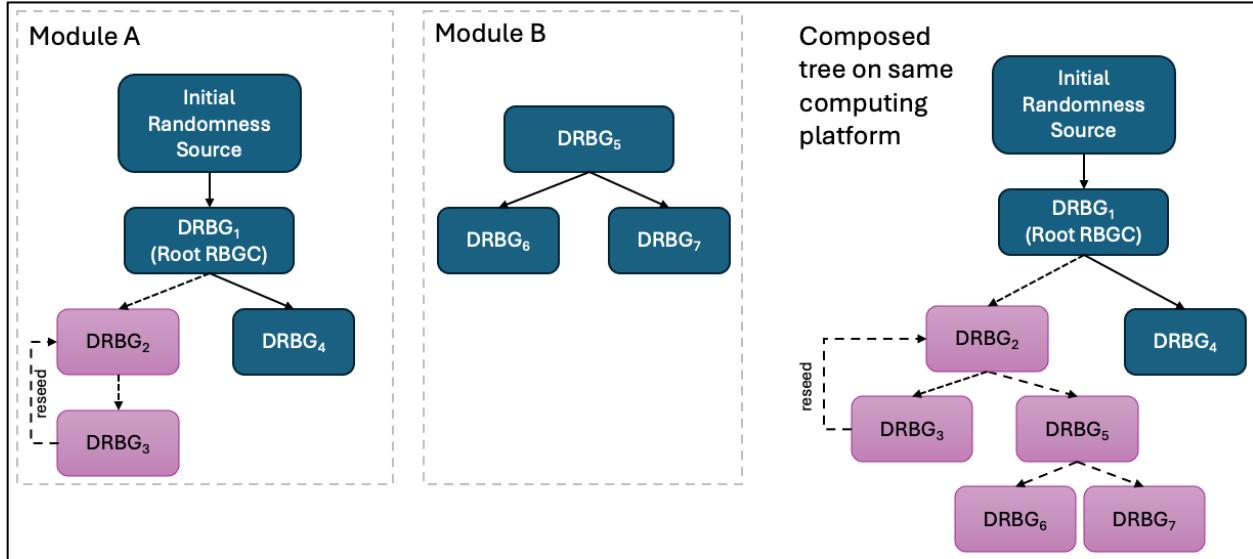


Fig. 39. Subtree in module B seeded by DRBG₂ of module A

3100 A.3.2. Changes in the Tree Structure

3101 New RBGC subtrees may be added to the tree during operation, and others may be removed. An
3102 RBGC construction may not be moved from one physical platform to another by any means,
3103 including backups, snapshots, and cloning.

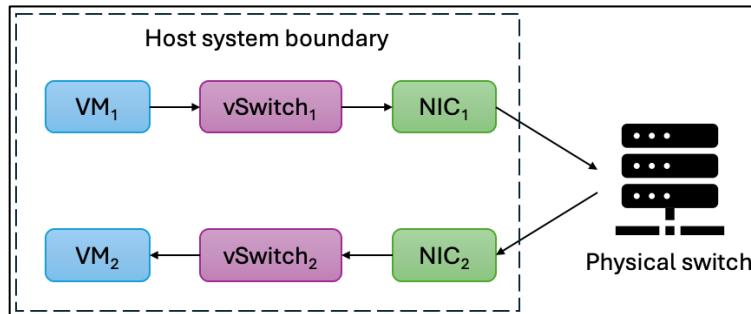
3104 An RBGC construction could be copied via forking within a single computer platform. Such cases
3105 are permissible as long as the original and/or new processes are reseeded prior to fulfilling any
3106 requests. This ensures that multiple instances of the same RBGC construction are not operating
3107 simultaneously with the same internal states. Without this reseeding, the outputs of one RBGC
3108 construction could be used to learn subsequent outputs from its counterpart, voiding any claims
3109 of prediction resistance.

3110 A.3.3. Using Virtual Machines

3111 The phrase “same computing platform” (used in Sec. 7) is intended to restrict realizations of RBGC
3112 constructions to similar concepts of a randomness source and DRBGs that exist within the same
3113 RBG boundary. In particular, seed material must pass from a randomness source to a DRBG in a
3114 way that provides the same guarantees as using a physical secure channel.

3115 RBGC constructions used within virtual machines (VMs) pose a unique challenge because they
3116 can be on the same physical platform yet communicate through a local area network (LAN).
3117 Whether network traffic between VMs is routed solely by the hypervisor’s virtual LAN (VLAN) or
3118 is sent to the platform’s network for routing depends on the configuration of the VLAN. For

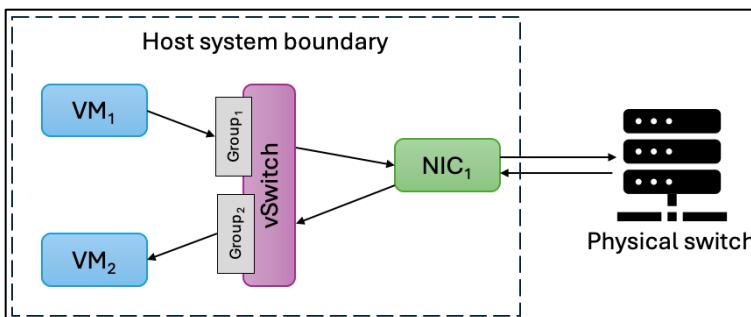
3119 example, two VMs that are in different port groups or use different virtual switches may transmit
3120 the data outside of the physical system they reside on, as shown in Fig. 40 and 41.



3121

3122

Fig. 40. VM_1 and VM_2 with different virtual switches

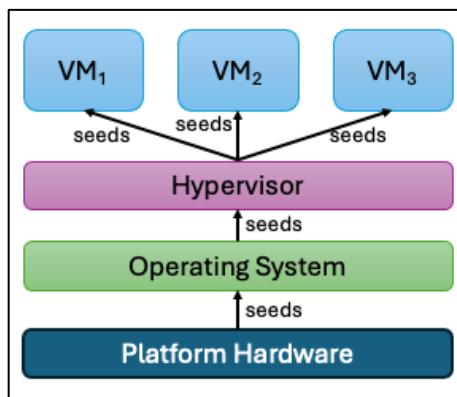


3123

3124

Fig. 41. VM_1 and VM_2 with the same virtual switch but different port groups

3125 A DRBG within a virtual machine could potentially obtain seed material from sources outside of
3126 the virtual machine if the seed material originates on the same computing platform. In particular,
3127 seed material can be obtained from randomness sources that reside in levels below the virtual
3128 machine, such as a hypervisor, host operating system, or the platform hardware. Figure 42 shows
3129 an example in which all seed material is obtained from lower levels on the same system.

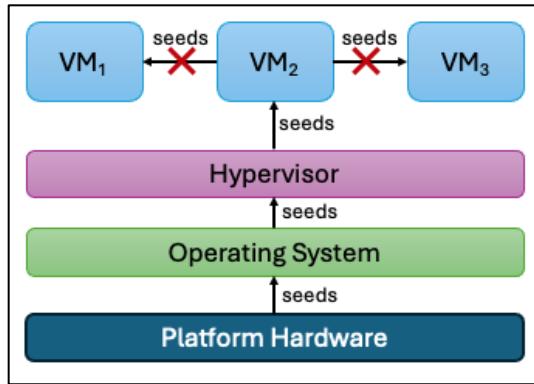


3130

3131

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

3132 To comply with an RBGC tree as specified in SP 800-90C, virtual machines cannot provide seed
3133 material to each other via a virtual network (see Fig. 43).



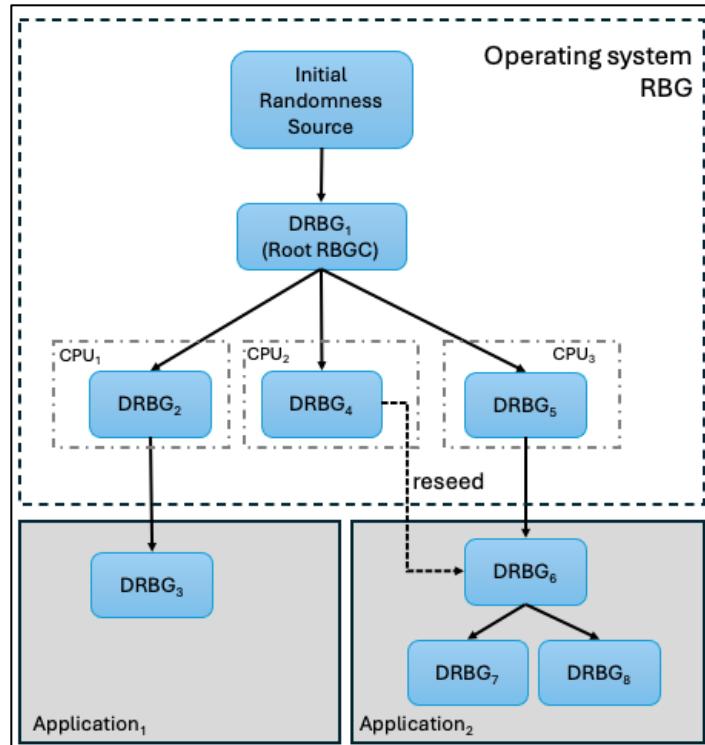
3134

3135 **Fig. 43. Acceptable external seeding for an RBGC construction in VM₂ but not in VM₁ and VM₃**

3136 This is a very important point in terms of local security guarantees. Virtual network configurations
3137 may change without being visible to a VM and alter the path of virtual network traffic. Therefore,
3138 it cannot be guaranteed that the seed material will never cross the physical network. Two
3139 configuration examples where data transmitted between virtual machines exits the host machine
3140 are shown in Fig. 40 and 41.

3141 **A.3.4. Reseeding From Siblings of the Parent**

3142 There may be situations in which it is acceptable for an RBGC construction to obtain reseeding
3143 material from an RBGC construction other than its parent. Figure 44 presents an example of a
3144 computing platform with an OS-level RBGC construction and tree containing an initial
3145 randomness source, root RBGC construction (containing DRBG₁), and three child RBGC
3146 constructions, each associated with a different processor (shown as CPU₁, CPU₂, and CPU₃).



3147

3148

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

3149 The DRBGs associated with these CPUs are DRBG₂, DRBG₃, and DRBG₄, each of which can be used
3150 as a randomness source by application-level RBGC constructions. Application₂ contains a subtree
3151 of RBGC constructions with DRBG₆, DRBG₇, and DRBG₈. This subtree is composed of the OS-level
3152 RBGC at DRBG₅ (i.e., DRBG₅ is the parent of DRBG₆).

3153 Ideally, DRBG₆ would obtain bits for reseeding from its parent, DRBG₅, but there may be reasons
3154 why this is either undesirable (e.g., because of load balancing) or not allowed by the RBGC
3155 requirements (e.g., seed material would exit the computing platform). Figure 44 provides an
3156 example in which a computing platform is a multi-processor system that performs load balancing
3157 to distribute tasks across processors. Application 2 (containing DRBG₆) was originally located on
3158 CPU₃ so that DRBG₆ was originally seeded by DRBG₅ (i.e., DRBG₅ is the parent of DRBG₆). If
3159 Application 2 is later moved to CPU₂ and DRBG₆ needs to be reseeded, it may be costly to reseed
3160 using DRBG₅. For efficiency within the multi-processor system, DRBG₆ can instead be reseeded
3161 using DRBG₄ if DRBG₄ has been designed and validated to meet the RBGC requirements. Note
3162 that DRBG₄ and DRBG₅ are siblings since they have the same parent (DRBG₁).

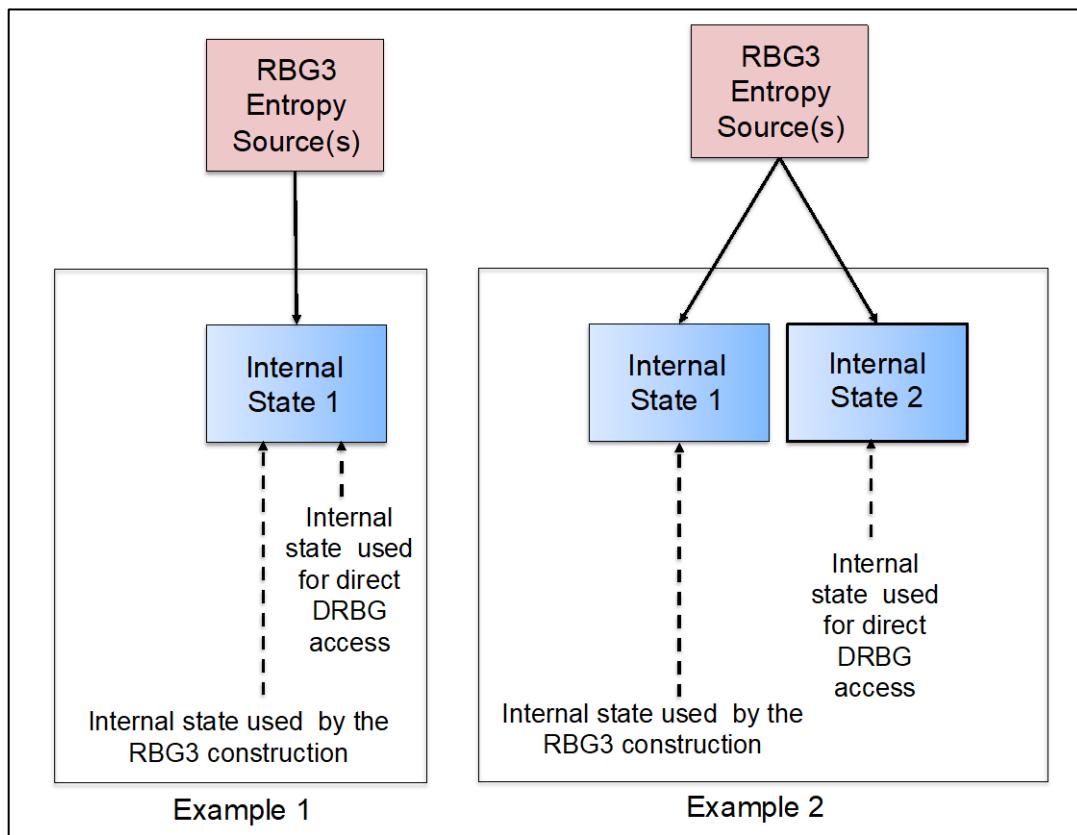
3163 **Appendix B. RBG Examples (Informative)**

3164 Appendix B.1 discusses and provides an example of the direct access to a DRBG used by an RBG3
3165 construction. Appendices B.2 – B.7 provide examples of each RBG construction.

3166 The figures do not show that if an error indicates an RBG failure (e.g., a noise source in the
3167 entropy source has failed), the RBG operation is terminated (see Sec. 2.6 and 8.1.2.1). For the
3168 examples below, all entropy sources are considered to be physical entropy sources. In order to
3169 simplify the examples, the *additional_input* parameter in the generate and reseed requests and
3170 generate functions is not used.

3171 **B.1. Direct DRBG Access in an RBG3 Construction**

3172 An implementation of an RBG3 construction may be designed so that the DRBG implementation
3173 used within the construction can be directly accessed by a consuming application using the same
3174 or separate instantiations from the instantiation used by the RBG3 construction (see the
3175 examples in Fig. 45).



3176

3177

Fig. 45. DRBG Instantiations

3178 In the leftmost example in Fig. 45, the same internal state is used by the RBG3 construction and
3179 a directly accessible DRBG. The DRBG implementation is instantiated only once, and only a single
3180 state handle is obtained during instantiation (e.g., *RBG3_DRBG_state_handle*). Generation and

3181 reseeding for RBG3 operations use RBG3 function calls (see Sec. 6.4 and 6.5), while generation
3182 and reseeding for direct DRBG access use RBG2 function calls (see Sec. 5.2) with the
3183 *RBG3_DRBG_state_handle*.

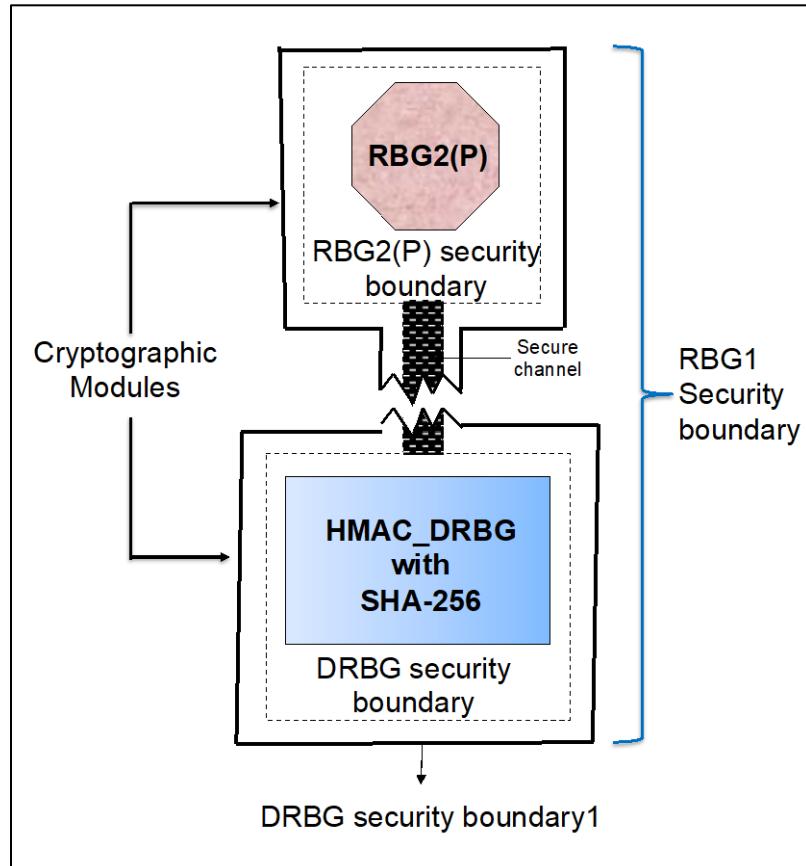
3184 In the rightmost example in Fig. 45, the RBG3 construction and directly accessible DRBG use
3185 different internal states. The DRBG implementation is instantiated twice — once for RBG3
3186 operations and a second time for direct access to the DRBG. A different state handle needs to be
3187 obtained for each instantiation (e.g., *RBG3_state_handle* and *RBG2_DRBG_state_handle*).
3188 Generation and reseeding for RBG3 operations use RBG3 function calls and
3189 *RBG3_DRBG_state_handle* (see Sec. 6.4 and 6.5), while generation and reseeding for direct
3190 DRBG access use RBG2 function calls and *RBG2_DRBG_state_handle* (see Sec. 5.2).

3191 Multiple directly accessible DRBGs may also be incorporated into an implementation by creating
3192 multiple instantiations. However, no more than one directly accessible DRBG should share the
3193 same internal state with the RBG3 construction (i.e., if n directly accessible DRBGs are required,
3194 either n or $n - 1$ separate instantiations are required).

3195 The directly accessed DRBG instantiations are in the same security boundary as the RBG3
3196 construction. When accessed directly using the same internal state as the RBG3 construction
3197 (rather than operating as part of the RBG3 construction), the DRBG operates as an RBG2(P)
3198 construction. A DRBG instantiation using a different internal state than the DRBG used by the
3199 RBG3 construction may operate as either an RBG2(P) or RBG2(NP) construction.

3200 **B.2. Example of an RBG1 Construction**

3201 An RBG1 construction only has access to a randomness source during instantiation (i.e., when it
3202 is seeded; see Sec. 4). In Fig. 46, the DRBG used by the RBG1 construction and the randomness
3203 source reside in two different cryptographic modules with a physically secure channel connecting
3204 them during the instantiation process.



3205

3206

Fig. 46. Example of an RBG1 construction

3207 Following DRBG instantiation, the secure channel is no longer available. For this example, the
3208 randomness source is an RBG2(P) construction (see Sec. 5) with a state handle of
3209 *RBG2_DRBG_state_handle*. The targeted security strength for the RBG1 construction is 256 bits,
3210 so a DRBG from SP 800-90A that is able to support this security strength must be used.
3211 HMAC_DRBG using SHA-256 is used in the example. A *personalization_string* is provided during
3212 instantiation, as recommended in Sec. 2.4.1.

3213 As discussed in Sec. 4, the randomness source (i.e., the RBG2(P) construction in this example) is
3214 not available during normal operation, so reseeding cannot be provided.

3215 This example provides an RBG that is instantiated at a security strength of 256 bits.

3216 **B.2.1. Instantiation of the RBG1 Construction**

3217 A physically secure channel is required to transport the entropy bits from the randomness source
3218 (i.e., the RBG2(P) construction) to the HMAC_DRBG during instantiation; an example of an
3219 RBG2(P) construction is provided in Appendix B.4. After the instantiation of the RBG1
3220 construction, the randomness source and the secure channel are no longer available.

3221 1. The HMAC_DRBG is instantiated by an application when sending an instantiate request
3222 to the DRBG:

3225 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”.

3229 2. The **DRBG_Instantiate_request** results in the execution of the **DRBG_Instantiate**
3230 function within the DRBG of the RBG1 construction (see Sec. 2.8.1.1):

3231 $(status, RBG1 \text{ } DRBG \text{ state handle}) = \text{DRBG_Instantiate}(256, \text{“Device 7056”}).$

3232 3. The instantiate function sends a reseed request to the RBG2(P) construction (i.e., the
3233 randomness source; see requirement 18 in Sec. 4.4.1).

3234 *status = DRBG Reseed request(RBG2 DRBG state handle),*

3235 where *RBG2_DRBG_state_handle* is the state handle for the internal state in the RBG2(P)
3236 construction.

3237 4. Upon receiving a reseed request, the RBG2(P) implementation executes a reseed
3238 function:

3239 *status = DRBG_Reseed(RBG2_DRBG state handle).*

3240 If an error is indicated by the returned *status*, the error is returned to the RBG1
3241 construction by the RBG2(P) construction in response to the reseed request and
3242 forwarded to the application by the RBG1 construction in response to the instantiate
3243 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).

3247 5. Upon receiving a *status* of success in response to the reseed request, the RBG1
3248 construction then sends a generate request to the RBG2(P) construction (see Sec. 5.2.2).

3251 where 384 is the 3s/2 bits needed to instantiate the HMAC_DRBG at a security strength
3252 of 256 bits.

3253 6. Upon receiving a generate request, the RBG2(P) construction executes a generate
3254 function using information from the request:

3255 $(status, seed\ material) = \text{DRBG Generate}(RBG2\ DRBG\ state\ handle, 384, 256)$.

3256 If an error is indicated by the returned *status*, the error is returned to the RBG1
3257 construction by the RBG2(P) construction in response to the generate request and
3258 forwarded to the application by the RBG1 construction in response to the instantiate
3259 request. The DRBG within the RBG1 construction is NOT instantiated.
3260 If a *status* of success is returned from the generate function, 384 bits of *seed_material* are
3261 also provided and sent to the RBG1 construction in response to the generate request.
3262 7. The DRBG within the RBG1 construction uses the *seed_material* provided by the RBG2(P)
3263 construction and the *personalization_string* provided by the application in the instantiate
3264 request (see step 1) to create the seed to instantiate the DRBG (see SP 800-90A).
3265 If the instantiation is not successful, an error is returned to the application in response to
3266 the instantiate request. The DRBG within the RBG1 construction has NOT been
3267 instantiated.
3268 If the instantiation is successful, the internal state is established. A *status* of SUCCESS and
3269 the *RBG1_DRBG_state_handle* are returned to the application requesting instantiation,
3270 and the RBG can be used to generate pseudorandom bits.

3271 **B.2.2. Generation by the RBG1 Construction**

3272 Assuming that the HMAC_DRBG in the RBG1 construction has been instantiated (see Appendix
3273 B.2.1), pseudorandom bits can be obtained as follows:

3274 1. A consuming application sends a generate request to the RBG1 construction:

3275 $(status, returned_bits) = \text{DRBG_Generate_request}(RBG1_DRBG_state_handle,$
3276 $requested_number_of_bits, requested_security_strength)$.

- 3277 • *RBG1_DRBG_state_handle* is returned as the state handle during instantiation
3278 (see Appendix B.2.1).
3279 • The *requested_security_strength* may be any value that is less than or equal to 256
3280 (i.e., the instantiated security strength recorded in the DRBG's internal state).

3281 2. Upon receiving a generate request, the RBG1 construction executes a generate function,
3282 as specified in Sec. 2.8.1.2:

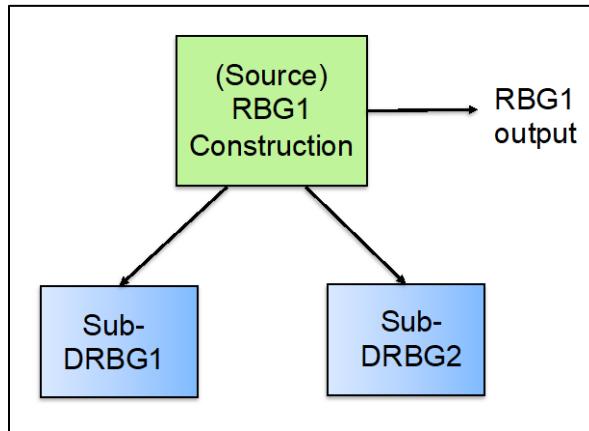
3283 $(status, returned_bits) = \text{DRBG_Generate}(RBG1_DRBG_state_handle,$
3284 $requested_number_of_bits, requested_security_strength)$.

3285 If an error is returned as the *status*, the RBG1 construction forwards the error indication
3286 to the application (in response to the generate request). *returned_bits* is a Null string.

3287 If an indication of success is returned as the *status*, the *requested_number_of_bits* are
3288 provided as the *returned_bits* to the consuming application in response to the generate
3289 request.

3290 **B.3. Example Using Sub-DRBGs Based on an RBG1 Construction**

3291 This example uses an RBG1 construction to instantiate two sub-DRBGs: sub-DRBG1 and sub-
3292 DRBG2 (see Fig. 47).



3293

3294 **Fig. 47. Sub-DRBGs based on an RBG1 construction**

3295 The instantiation of the RBG1 construction is discussed in Appendix B.2. The RBG1 construction
3296 that is used as the randomness source includes an HMAC_DRBG and has been instantiated to
3297 provide a security strength of 256 bits. The state handle for the construction is
3298 *RBG1_DRBG_state_handle*.

3299 For this example, sub-DRBG1 will be instantiated to provide a security strength of 128 bits, and
3300 sub-DRBG2 will be instantiated to provide a security strength of 256 bits. Both sub-DRBGs use
3301 the same DRBG algorithm as the RBG1 construction (i.e., HMAC_DRBG using SHA-256). Neither
3302 the RBG1 construction nor the sub-DRBGs can be reseeded.

3303 This example provides the following capabilities:

- 3304 • Access to the RBG1 construction to provide output generated at a security strength of
3305 256 bits (see Appendix B.2 for the RBG1 example),
- 3306 • Access to one sub-DRBG (i.e., sub-DRBG1) that provides output for an application that
3307 requires a security strength of no more than 128 bits, and
- 3308 • Access to a second sub-DRBG (i.e., sub-DRBG2) that provides output for a second
3309 application that requires a security strength of 256 bits.

3310 **B.3.1. Instantiation of the Sub-DRBGs**

3311 Each sub-DRBG is instantiated using output from an RBG1 construction that is discussed in
3312 Appendix B.2.

3313 **B.3.1.1. Instantiating Sub-DRBG1**

- 3314 1. Sub-DRBG1 is instantiated when an application sends an instantiate request to the RBG1
3315 construction:

3316 $(status, sub\text{-}DRBG1_state_handle) =$
3317 **Instantiate_sub\text{-}DRBG_request(128, "Sub\text{-}DRBG App 1")**,

3318 where

- 3319 • A security strength of 128 bits is requested for sub-DRBG1,
- 3320 • The *personalization string* to be used for sub-DRBG1 is "Sub-DRBG App 1", The
3321 comma is Nand
- 3322 • The returned state handle for sub-DRBG1 will be *sub-DRBG1_state_handle*.

- 3323 2. Upon receiving the instantiate request, the RBG1 construction executes its instantiate
3324 function for a sub-DRBG (see Sec. 4.3.1):

3325 $(status, sub\text{-}DRBG1_state_handle) = \mathbf{Instantiate_sub\text{-}DRBG}(128,$
3326 "Sub-DRBG App 1").

3327 As specified for the **Instantiate_sub\text{-}DRBG** function, the DRBG in the RBG1 construction
3328 will attempt to generate $3s/2 = 192$ bits of seed material and combine it with "Sub-DRBG
3329 App 1" (i.e., the personalization string) to create a seed for the internal state of sub-
3330 DRBG1.

3331 If an error is returned as the *status*, the RBG1 construction forwards the error indication
3332 to the application in response to the instantiate request. The sub-DRBG is NOT
3333 instantiated.

3334 If an indication of success is returned as the *status*, the RBG1 construction forwards the
3335 *status* to the application in response to the instantiate request. Sub-DRBG1 can now be
3336 requested directly to generate output. See Appendix B.3.2.

3337 **B.3.1.2. Instantiating Sub-DRBG2**

3338 Sub-DRBG2 is instantiated in the same manner as sub-DRBG1 but at a security strength of 256
3339 bits and with a different personalization string.

- 3340 1. The application sends an instantiate request to the RBG1 construction:

3341 $(status, sub\text{-}DRBG2_state_handle) =$
3342 **Instantiate_sub\text{-}DRBG_request(256, "Sub\text{-}DRBG App 2")**.

- 3343 2. The RBG1 construction executes an instantiate function for a sub-DRBG:

3344 $(status, sub\text{-}DRBG2_state_handle) = \mathbf{Instantiate_sub\text{-}DRBG}(256,$
3345 "Sub-DRBG App 2").

3346 The DRBG in the RBG1 construction will attempt to generate $3s/2 = 384$ bits of seed
3347 material and combine it with “Sub-DRBG App 2” to create a seed for the internal state of
3348 sub-DRBG2.

3349 If an error is returned as the *status*, the RBG1 construction forwards the error indication
3350 to the application in response to the instantiate request. The sub-DRBG is NOT
3351 instantiated.

3352 If an indication of success is returned as the *status*, the RBG1 construction forwards the
3353 *status* to the application in response to the instantiate request. Sub-DRBG2 can now be
3354 requested directly to generate output. See Appendix B.3.2.

3355 **B.3.2. Pseudorandom Bit Generation by Sub-DRBGs**

3356 Assuming that the sub-DRBG has been successfully instantiated (see Appendix B.3.1),
3357 pseudorandom bits can be requested from the sub-DRBG by a consuming application.

3358 1. An application sends the following generate request:

3359 $(status, returned_bits) = \text{DRBG_Generate_request}(sub\text{-}DRBG\text{_state__handle},$
3360 $requested\text{_number__of__bits}, requested\text{_security__strength}),$

- 3361 • For sub_DRBG1, *sub-DRBG_state_handle* = *sub-DRBG1_state_handle*.
3362 • For sub-DRBG2, *sub-DRBG_state_handle* = *sub-DRBG2_state_handle*.
3363 • *requested_number_of_bits* must be $\leq 2^{19}$ (see SP 800-90A for the HMAC_DRBG
3364 parameters).
3365 • For sub_DRBG1, security strength must be ≤ 128 .
3366 • For sub_DRBG2, security strength must be ≤ 256 .

3367 2. The sub-DRBG executes the generate request (see Sec. 2.8.1.2):

3368 $(status, returned_bits) = \text{DRBG_Generate}(sub\text{-}DRBG\text{_state__handle},$
3369 $requested\text{_number__of__bits}, security\text{_strength}).$

3370 If an error is returned as the *status*, the sub-DRBG forwards the error indication to the
3371 application in response to the generate request. The *returned_bits* string is *Null*.

3372 If an indication of success is returned as the *status*, the sub-DRBG forwards the *status* to
3373 the application along with the requested number of newly generated bits.

3374 **B.4. Example of an RBG2(P) Construction**

3375 For this example of an RBG2(P) construction, no conditioning function is used, and only a single
3376 DRBG instantiation will be used (see Fig. 48), so a state handle is not needed. A physical and a
3377 non-physical entropy source are used. Full-entropy output is not provided by the entropy
3378 sources.

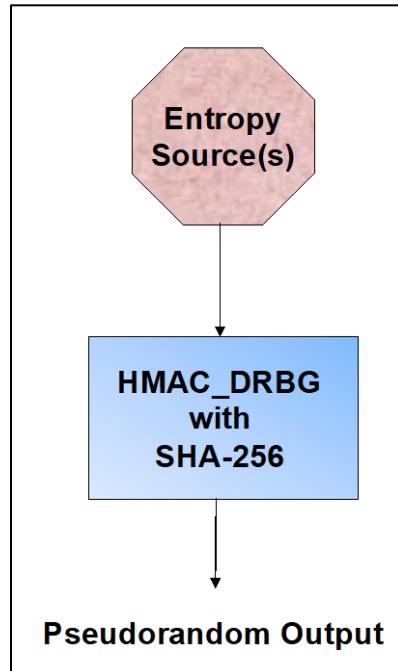


Fig. 48. Example of an RBG2 construction

3379

3380

3381 The targeted security strength is 256 bits, so a DRBG from SP 800-90A that can support this
3382 security strength must be used; HMAC_DRBG using SHA-256 is used in this example. A
3383 *personalization_string* may be provided, as recommended in Sec. 2.4.1. Reseeding is supported
3384 and will be available on demand. Method 1 is used for counting the entropy produced by the
3385 entropy sources (i.e., only entropy from the physical entropy source is counted).

3386 This example provides the following capabilities:

- 3387
 - An RBG instantiated at a security strength of 256 bits and
 - Access to an entropy source to provide prediction resistance.

3389 **B.4.1. Instantiation of an RBG2(P) Construction**

3390 1. The RBG2(P) construction is instantiated by an application using an instantiate request:

3391 $status = \text{DRBG_Instantiate_request}(256, "RBG2\ 42")$.

3392 Since there is only a single instantiation, a *state_handle* is not used for this example. The
3393 *personalization_string* to be used for this example is “RBG2 42”.

3394 2. Upon receiving the instantiate request, the RBG2(P) construction executes an instantiate
3395 function:

3396 $status = \text{DRBG_Instantiate}(256, "RBG2\ 42")$.

3397 The seed material for establishing the security strength (s) of the DRBG (i.e., $s = 256$ bits)
3398 is requested using the following call to the entropy source (see Sec. 2.8.2 and item 2 in
3399 Sec. 5.2.1):

3400 $(status, seed_material) = \text{Get_entropy_bitstring}(384, Method_1),$

3401 where $3s/2 = 384$ bits of entropy are requested from the entropy source, and Method 1
3402 is used to count only the entropy produced by the physical entropy source.

3403 If $status = \text{SUCCESS}$ is returned in response to the **Get_entropy_bitstring** call, the
3404 HMAC_DRBG is seeded using *seed_material* and the *personalization_string* ("RBG2
3405 42"). The internal state is recorded (including the security strength of the instantiation),
3406 and $status = \text{SUCCESS}$ is returned to the consuming application in response to the
3407 instantiation request.

3408 If the *status* returned in response to the **Get_entropy_bitstring** call indicates an error,
3409 then the internal state is not created, the *status* is returned to the consuming application
3410 in response to the instantiation request, and the RBG cannot be used to generate bits.

3411 **B.4.2. Generation Using an RBG2(P) Construction**

3412 Assuming that the RBG has been successfully instantiated (see Appendix B.4.1):

3413 1. Pseudorandom bits can be requested from the RBG by a consuming application:

3414 $(status, returned_bits) = \text{DRBG_Generate_request}(requested_number_of_bits,$
3415 *requested_security_strength*).

- 3416 • Since there is only a single instantiation of the HMAC_DRBG, a *state_handle* was
3417 not returned from the **DRBG_Instantiate** (see Appendix B.4.1) and is not used
3418 during the generate request.
- 3419 • The *requested_security_strength* may be any value that is ≤ 256 (i.e., the
3420 instantiated security strength recorded in the HMAC_DRBG's internal state).

3421 2. Upon receiving the generate request, the RBG executes the generate function (see Sec.
3422 2.8.1.2):

3423 $(status, returned_bits) = \text{DRBG_Generate}(requested_number_of_bits,$
3424 *security_strength*).

3425 A *status* indication is returned to the requesting application in response to the
3426 **DRBG_Generate** call. If $status = \text{SUCCESS}$, a bitstring of at least
3427 *requested_number_of_bits* is provided as the *returned_bits*. If $status = \text{FAILURE}$,
3428 *returned_bits* is an empty bitstring.

3429 **B.4.3. Reseeding an RBG2(P) Construction**

3430 The HMAC_DRBG will be reseeded 1) if explicitly requested by the consuming application or 2)
3431 automatically during a **DRBG_Generate** call at the end of the DRBG's designed *seedlife* (see the
3432 **DRBG_Generate** function specification in SP 800-90A and Sec. 5.2.3 herein).

3433 1. An application may request a reseed of the DRBG using a reseed request:

3434 $status = \text{DRBG_Reseed_request}()$.

3435 Since there is only a single instantiation of the HMAC_DRBG, a *state_handle* was not
3436 returned from the **DRBG_Instantiate** function (see Appendix B.4.1) and is not used
3437 during the reseed request.

3438 2. Upon receiving the reseed request or when the end of the seedlife is determined, the RBG
3439 executes the reseed function (see Sec. 2.8.1.3):

3440 $status = \text{DRBG_Reseed}()$.

3441 The **DRBG_Reseed** function uses a **Get_randomness-source_input** call to access the
3442 entropy source.

3443 $(status, seed_material) = \text{Get_entropy_bitstring}(256, Method_I)$.

3444 *Method_I* indicates that only the entropy from the physical entropy source should be
3445 counted.

3446 If *status* = SUCCESS is returned by **Get_entropy_bitstring**, the *seed_material* contains
3447 at least 256 bits of entropy and is at least 256 bits long. *Status* = SUCCESS is returned to
3448 the RBG2 construction in response to the **DRBG_Reseed** call, and the *status* is forwarded
3449 to the application in response to the reseed request, if appropriate.

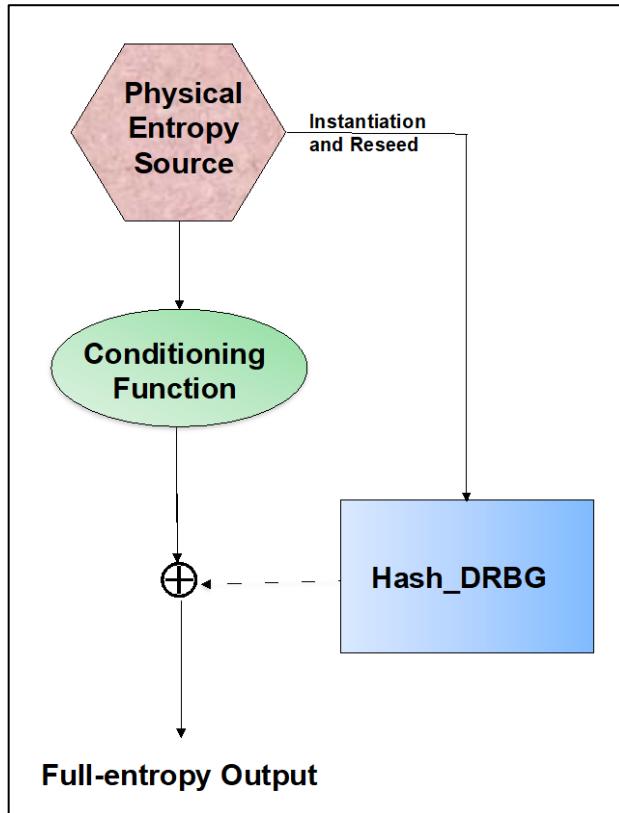
3450 If the *status* indicates an error, *seed_material* is an empty (e.g., null) bitstring. The
3451 HMAC_DRBG is not reseeded, the *status* is returned to the **DRBG_Reseed** function in
3452 the RBG2 construction, and the *status* is then forwarded to the application in response to
3453 the reseed request, if appropriate. Depending on the error, the DRBG operation may be
3454 terminated (see item 10 in Sec. 2.6).

3455 **B.5. Example of an RBG3(XOR) Construction**

3456 This construction is specified in Sec. 6.4 and requires a DRBG and a source of full-entropy bits.
3457 For this example, a single physical entropy source that does not provide full-entropy output is
3458 used, so the vetted hash conditioning function listed in SP 800-90B using SHA-256 is used as an
3459 external conditioning function. Since the type of entropy source is known, the counting method
3460 is known and need not be indicated when requesting entropy.

3461 The Hash_DRBG specified in SP 800-90A will be used as the DRBG with SHA-256 used as the
3462 underlying hash function for the DRBG (note the use of SHA-256 for both the Hash_DRBG and
3463 the vetted conditioning function). The DRBG will obtain input directly from the RBG's entropy

3464 source without conditioning (as shown in Fig. 49) since bits with full entropy are not required for
3465 input to the DRBG, even though full-entropy bits are required for input to the XOR operation
3466 (shown as “ \oplus ” in the figure) from the entropy source via the conditioning function.



3467

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

3469 The DRBG is instantiated and reseeded at a 256-bit security strength. In this example, only a
3470 single instantiation is used, and a personalization string is provided during instantiation. Calls are
3471 made to the RBG using the RBG3(XOR) calls specified in Sec. 6.4. The Hash_DRBG itself is not
3472 directly accessible.

3473 This example provides the following capabilities:

- 3474 • Full-entropy output by the RBG,
3475 • Fallback to the security strength provided by the Hash_DRBG (256 bits) if the entropy
3476 source has an undetected failure, and
3477 • Access to an entropy source to instantiate and reseed the Hash_DRBG.

3478 B.5.1. Instantiation of an RBG3(XOR) Construction

- 3479 1. An application instantiates an RBG3(XOR) construction using an instantiate request that
3480 will instantiate the DRBG within the RBG:

3481 $status = \text{Instantiate_RBG3_DRBG_request}(256, "RBG3(XOR)").$

3482 Since only a single instantiation is used, there is no need for a state handle. The
3483 HMAC_DRBG is requested to be instantiated at a security strength of 256 bits using
3484 "RBG3(XOR)" as a personalization string.

3485 2. Upon receiving an instantiate request, the RBG3(XOR) construction executes an
3486 instantiate function:

3487 $status = \text{RBG3(XOR)_Instantiate}(256, "RBG3(XOR)").$

3488 The entropy for establishing the security strength (s) of the Hash_DRBG (i.e., where $s =$
3489 256 bits) is requested from the entropy source using the following
3490 **Get_entropy_bitstring** call:

3491 $(status, seed_material) = \text{Get_entropy_bitstring}(384).$

3492 If $status = \text{SUCCESS}$ is returned from the **Get_entropy_bitstring** call, the Hash_DRBG
3493 is seeded using the *seed_material* and the *personalization_string* (i.e., "RBG3(XOR)"). The
3494 internal state is recorded (including the 256-bit security strength of the instantiation), and
3495 $status = \text{SUCCESS}$ is returned to the RBG3(XOR) construction and forwarded to the
3496 consuming application in response to the instantiate request (from step 1). The RBG can
3497 be used to generate full-entropy bits.

3498 If the $status$ returned from the **Get_entropy_bitstring** call indicates an error, the $status$
3499 is forwarded by the RBG3(XOR) construction to the consuming application. The
3500 Hash_DRBG's internal state is not established, and the RBG cannot be used to generate
3501 bits.

3502 B.5.2. Generation by an RBG3(XOR) Construction

3503 Assuming that the Hash_DRBG has been instantiated (see Appendix B.5.1), the RBG can be
3504 called by a consuming application to generate output with full entropy.

3505 B.5.2.1. Generation

3506 1. An application requests the generation of full-entropy bits using:

3507 $(status, returned_bits) = \text{RBG3_DRBG_Generate_request}(n),$

3508 where n indicates the requested number of bits to generate. A state handle is not included
3509 since a state handle was not returned during instantiation (see Appendix B.5.1).

3510 2. Upon receiving a generate request, the RBG3(XOR) construction executes a call to the
3511 generate function:

3512 $(status, returned_bits) = \text{RBG3(XOR)_Generate}(n).$

3513 The construction of the **RBG3(XOR)_Generate** function in Sec. 6.4.1.2 is used as
3514 follows:

3515 **RBG3(XOR)_Generate:**

3516 **Input:**

3517 *n*: The number of bits to be generated.

3518 **Output:**

3519 *status*: The status returned by the **RBG3(XOR)_Generate** function.

3520 *returned_bits*: The newly generated bits or a *Null* bitstring.

3521 **Process:**

3522 2.1 (*status*, *ES_bits*) = **Get_conditioned_full-entropy_input**(*n*).

3523 2.2 If (*status* ≠ SUCCESS), then return(*status*, *Null*).

3524 2.3 (*status*, *DRBG_bits*) = **DRBG_Generate**(*n*, 256).

3525 2.4 If (*status* ≠ SUCCESS), then return(*status*, *Null*).

3526 2.5 *returned_bits* = *ES_bits* ⊕ *DRBG_bits*.

3527 2.6 Return (SUCCESS, *returned_bits*).

3528 The *state_handle* parameter is not used in the **RBG3(XOR)_Generate** call or the
3529 **DRBG_Generate** function call (in step 2.3) for this example since a *state_handle* was not
3530 returned from the **RBG3(XOR)_Instantiate** function (see Appendix B.5.1).

3531 In step 2.1, the entropy source is accessed via the conditioning function using the
3532 **Get_conditioned_full-entropy_input** routine (see Appendix B.5.2.2) to obtain *n* bits with
3533 full entropy, which are returned as the *ES_bits*.

3534 Step 2.2 checks that the **Get_conditioned_full-entropy_input** call in step 2.1 was
3535 successful. If it was not successful, the **RBG3(XOR)_Generate** function is aborted,
3536 returning *status* ≠ SUCCESS and a *Null* bitstring to the RBG3(XOR) construction. The
3537 *status* and *Null* bitstring are then forwarded to the application in response to the generate
3538 request (in step 1).

3539 Step 2.3 calls the **Hash_DRBG** to generate *n* bits at a security strength of 256 bits. The
3540 generated bitstring is returned as *DRBG_bits*.

3541 Step 2.4 checks that the **DRBG_Generate** function invoked in step 2.3 was successful. If
3542 it was not successful, the **RBG3(XOR)_Generate** function is aborted, returning *status* ≠
3543 SUCCESS and a *Null* bitstring to the RBG3(XOR) construction. The *status* and *Null*
3544 bitstring are then forwarded to the application in response to the generate request (in
3545 step 1).

3546 If step 2.3 returns an indication of success, the *ES_bits* returned in step 2.1 and the
3547 *DRBG_bits* obtained in step 2.3 are XORed together in step 2.5. The result is returned to
3548 the RBG3(XOR) construction in step 2.6 and forwarded to the application in response to
3549 the generate request (in step 1).

3550 **B.5.2.2. Get_conditioned_full-entropy_input Function**

3551 The **Get_conditioned_full-entropy_input** procedure is specified in Sec. 3.2.2.2. For this
3552 example, the routine becomes the following:

3553 **Get_conditioned_full_entropy_input:**

3554 **Input:**

3555 n : The number of full-entropy bits to be provided.

3556 **Output:**

3557 1. $status$: The status returned from the **Get_conditioned_full_entropy_input** function.

3558 2. *Full-Entropy_bitstring*: The newly acquired n -bit string with full entropy or a *Null*
3559 bitstring.

3560 **Process:**

3561 1. $temp = \text{the Null string}$.

3562 2. $ctr = 0$.

3563 3. While $ctr < n$, do

3564 3.1 $(status, Entropy_bitstring) = \text{Get_entropy_bitstring}(320)$.

3565 3.2 If ($status \neq \text{SUCCESS}$), then return ($status, Null$).

3566 3.3 $conditioned_output = \text{HashSHA_256}(Entropy_bitstring)$.

3567 3.4 $temp = temp \parallel conditioned_output$.

3568 3.5 $ctr = ctr + 256$.

3569 4. *Full-Entropy_bitstring* = **leftmost**($temp, n$).

3570 5. Return ($\text{SUCCESS}, Full-Entropy_bitstring$).

3571 Steps 1 and 2 initialize the temporary bitstring ($temp$) for holding the full-entropy bitstring being
3572 assembled and the counter (ctr) that counts the number of full-entropy bits produced so far.

3573 Step 3 obtains and processes the entropy for each iteration.

3574 • Step 3.1 requests 320 bits from the entropy source (i.e., $output_len + 64$ bits, where
3575 $output_len = 256$ for SHA-256).

3576 • Step 3.2 checks whether the $status$ returned in step 3.1 indicated a success. If the $status$
3577 did not indicate a success, the $status$ is returned to the **RBG3(XOR)_Generate** function
3578 (in Appendix B.5.2.1) along with a *Null* bitstring.

3579 • Step 3.3 invokes the hash conditioning function (see Sec. 3.2.1.2) using SHA-256 for
3580 processing the *Entropy_bitstring* obtained from step 3.1.

- 3581 • Step 3.4 concatenates the *conditioned_output* received in step 3.3 to the temporary
3582 bitstring (*temp*).
3583 • Step 3.5 increments the counter for the number of full-entropy bits that have been
3584 produced so far.
3585 After at least n bits have been produced in step 3, step 4 selects the leftmost n bits of the
3586 temporary string (*temp*) to be returned as the bitstring with full entropy.
3587 Step 5 returns the result from step 4 (i.e., *Full-Entropy_bitstring*).

3588 **B.5.3. Reseeding an RBG3(XOR) Construction**

3589 The Hash_DRBG within the RBG3(XOR) construction must be reseeded at the end of its designed
3590 seedlife and may be reseeded on demand (e.g., by the consuming application). Reseeding will be
3591 automatic whenever the end of the DRBG's seedlife is reached during a **DRBG_Generate** call
3592 (see SP 800-90A and step 2.3 in Appendix B.5.2.1).

3593 The consuming application uses a reseed request to reseed the DRBG within the RBG3(XOR)
3594 construction:

3595 $status = \text{DRBG_Reseed_request}()$.

3596 A state handle is not provided for this example since none was provided during instantiation.

3597 Whether reseeding is done automatically during a **DRBG_Generate** call or is specifically
3598 requested by a consuming application, the **DRBG_Reseed** call for this example is:

3599 $status = \text{DRBG_Reseed}()$.

3600 Again, a state handle is not provided since none was provided during instantiation.

3601 A **Get_entropy_bitstring** call to the entropy source is used to obtain the entropy for reseeding:

3602 $(status, seed_material) = \text{Get_entropy_bitstring}(256)$.

3603 If $status = \text{SUCCESS}$ is returned by the **Get_entropy_bitstring** call, *seed_material* consists of at
3604 least 256 bits that contain at least 256 bits of entropy. These bits are used by the **DRBG_Reseed**
3605 function to reseed the Hash_DRBG. If the reseed was requested by an application, the *status* is
3606 returned to that application.

3607 If the *status* indicates an error, the *seed_material* is a *Null* bitstring, and the Hash_DRBG is not
3608 reseeded. If the reseed was requested by an application, the error *status* is returned to the
3609 application.

3610 **B.6. Example of an RBG3(RS) Construction**

3611 This construction is specified in Sec. 6.5 and requires an entropy source and a DRBG, which is
3612 shown in the left half of Fig. 50 outlined in green with long dashes (— — —). The DRBG is directly
3613 accessible using the same instantiation that is used by the RBG3(RS) construction (i.e., they share

3614 the same internal state). When accessed directly, the DRBG behaves as an RBG2(P) construction,
 3615 which is shown in the right half of Fig. 50 outlined in blue with alternating dots and dashes (- - - -).
 3616

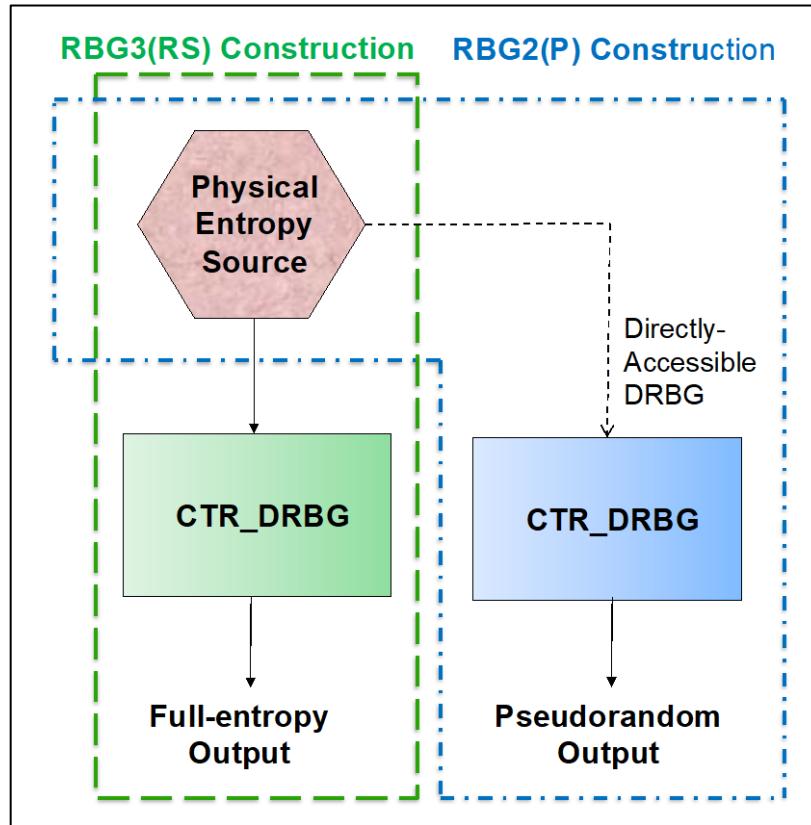


Fig. 50. Example of an RBG3(RS) construction

3617
 3618
 3619 The CTR_DRBG specified in SP 800-90A will be used as the DRBG with AES-256 used as the
 3620 underlying block cipher for the DRBG. The CTR_DRBG will be implemented using a derivation
 3621 function located inside of the CTR_DRBG implementation. In this case, full-entropy output will
 3622 not be required from the entropy source (see SP 800-90A).

3623 As specified in Sec. 6.5, a DRBG used as part of the RBG must be instantiated and reseeded at a
 3624 security strength of 256 bits when AES-256 is used in the DRBG.

3625 For this example, the DRBG has a fixed security strength (i.e., 256 bits), which is hard-coded into
 3626 the implementation so will not be used as an input parameter.

3627 Calls are made to the RBG3(RS) construction, as specified in Sec. 6.5. Calls made to the directly
 3628 accessible DRBG (part of the RBG2(P) construction) use the RBG calls specified in Sec. 5. Since an
 3629 entropy source is always available, the directly accessed DRBG can be reseeded.

3630 If the entropy source produces output at a slow rate, a consuming application might call the
 3631 RBG3(RS) construction only when full-entropy bits are required, obtaining all other output from
 3632 the directly accessible DRBG. Requirement 2 in Sec. 6.5.2 requires that the DRBG be reseeded

3633 whenever a request for generation by a directly accessible DRBG follows a request for generation
3634 by the RBG3(RS) construction. For this example, a global variable (*last_call*) within the RBG3(RS)
3635 security boundary is used to indicate whether the last use of the DRBG was as part of the
3636 RBG3(RS) construction or directly accessed:

- 3637 • *last_call* = 1 if the DRBG was last used as part of the RBG3(RS) construction to provide
3638 full entropy output. If the next request is for generation by the DRBG directly, the DRBG
3639 must be reseeded before the requested output is generated.
- 3640 • *last_call* = 0 otherwise. A reseed of the DRBG when accessed directly is not necessary.
3641 When the DRBG is first instantiated with entropy, *last_call* is set to zero.

3642 See SP 800-90Ar1 for information about the internal state of the CTR_DRBG.

3643 This example provides the following capabilities:

- 3644 • Full-entropy output by the RBG3(RS) construction,
- 3645 • Fallback to the security strength of the RBG3(RS)'s DRBG instantiation (i.e., 256 bits) if the
3646 entropy source has an undetected failure,
- 3647 • Direct access to the DRBG with a security strength of 256 bits for faster output when full-
3648 entropy output is not required,
- 3649 • Access to an entropy source to instantiate and reseed the DRBG, and
- 3650 • On-demand reseeding of the DRBG (e.g., to provide prediction resistance for requests to
3651 the directly accessed DRBG).

3652 **B.6.1. Instantiation of an RBG3(RS) Construction**

3653 Instantiation for this example consists of the instantiation of the CTR_DRBG used by the
3654 RBG3(RS) construction.

- 3655 1. An application requests the instantiation of the RBG3(RS) construction using:

3656 $(status, RBG3_DRBG_state_handle) = \text{Instantiate_RBG3_DRBG_request}(\text{"RBG3(RS)}\\$
3657 $\quad 2024")$,

3658 which requests the instantiation of the DRBG within the RBG3(RS) construction using
3659 "RBG3(RS) 2024" as the personalization string. In this example, the request does not
3660 include an indication of the security strength to be instantiated that would need to be
3661 checked against the security strength implemented for the DRBG (see Sec. 2.8.3.1 for a
3662 discussion).

- 3663 2. Upon receiving the request, the RBG3(RS) construction executes the instantiate function:

3664 $(status, RBG3_DRBG_state_handle) = \text{RBG3(RS)_Instantiate}(\text{"RBG3(RS) 2024"})$.

3665 For this example, the **RBG3(RS)_Instantiate** function (see Sec. 6.5.1.1) in the DRBG includes an
3666 additional step to set the initial value of *last_call* to zero (i.e., if the first use of the DRBG is for

3667 direct access, a reseed of the DRBG before generating bits is not required). Setting the initial
3668 value of *last_call* is an implementation decision, but some method for this process is required:

3669 2.1 $(status, RBG3_DRBG_state_handle) = \mathbf{DRBG_Instantiate}(personalization_string)$.

3670 2.2 $last_call = 0$.

3671 2.3 Return(*status*, *RBG3_DRBG_state_handle*).

3672 In step 2.1, the **DRBG_Instantiate** function is used to instantiate the CTR_DRBG using
3673 “RBG3(RS) 2024” as the personalization string. Since the required security strength is known (i.e.,
3674 256 bits) and a derivation function is used in the CTR_DRBG implementation, the required
3675 entropy ($s + 128 = 384$ bits) is obtained from the entropy source using:

3676 $(status, seed_material) = \mathbf{Get_entropy_bitstring}(s + 128)$.

3677 The *seed_material* and personalization string are used to seed the CTR_DRBG. Since the
3678 entropy source is known to be a physical entropy source, the counting method is known and not
3679 included as an input parameter.

3680 Step 2.2 sets *last_call* = 0 so that if the initial request is for direct access to the DRBG, a reseed
3681 will not be initially required before generating bits (i.e., entropy has just been acquired as a result
3682 of the instantiation process).

3683 In step 2.3, the *status* and the state handle for the DRBG’s internal state are returned to the
3684 **RBG3(RS)_Instantiate** function and forwarded to the application in response to the instantiate
3685 request in step 1.

3686 B.6.2. Generation by an RBG3(RS) Construction

3687 Assuming that the DRBG in the RBG3(RS) construction has been instantiated (see Appendix
3688 B.6.1), the RBG can be invoked by a consuming application to generate outputs with full entropy.

3689 1. An application requests the generation of full-entropy bits using:

3690 $(status, returned_bits) = \mathbf{RBG3_Generate_request}(RBG3_DRBG_state_handle, n)$,

3691 where *RBG3_DRBG_state_handle* was provided during DRBG instantiation (see Appendix B.6.1),
3692 and *n* is the number of requested bits.

3693 2. Upon receiving the generate request, the RBG3(RS) construction executes the generate
3694 function (see Sec. 6.5.1.2.1):

3695 $(status, returned_bits) = \mathbf{RBG3(RS)_Generate}(RBG3_DRBG_state_handle, n)$.

3696 A few modifications to the **RBG3(RS)_Generate** function have been made, resulting in
3697 the following:

3698 **RBG3(RS)_Generate:**

3699 **Input:**

- 3700 • *RBG3_DRBG_state_handle*: The state handle for the DRBG's internal state
3701 (see Appendix B.6.1).

- 3702 • *n*: The number of full-entropy bits to be generated.

3703 **Output:**

- 3704 • *status*: The status returned from the **RBG3(RS)_Generate** function.

- 3705 • *returned_bits*: The newly generated bits or a *Null* bitstring.

3706 **Process:**

3707 2.1 *temp* = *Null*.

3708 2.2 *sum* = 0.

3709 2.3 While (*sum* < *n*),

3710 2.3.1 *status* = **DRBG_Reseed**(*RBG3_DRBG_state_handle*).

3711 2.3.2 If (*status* ≠ SUCCESS), then return (*status*, *Null*).

3712 2.3.3 (*status*, *full_entropy_bits* =
3713 **DRBG_Generate**(*RBG3_DRBG_state_handle*, 256)).

3714 2.3.4 If (*status* ≠ SUCCESS), then return (*status*, *Null*).

3715 2.3.5 *temp* = *temp* || *full_entropy_bits*.

3716 2.3.6 *sum* = *sum* + *s*.

3717 2.4 *last_call* = 1.

3718 2.5 Return (SUCCESS, **leftmost**(*temp*, *n*)).

3719 Steps 2.1 and 2.2 initialize *temp* to a *Null* string for accumulating the requested output
3720 and *sum* to zero for counting the entropy generated.

3721 Step 2.3 generates the requested output with full entropy.

3722 Step 2.3.1 reseeds the DRBG. Whenever the RBG3(RS) construction is requested to
3723 generate bits, the DRBG is always reseeded with *s* + 64 = 320 bits directly from the
3724 entropy source (see Appendix B.6.4).

3725 Step 2.3.2 checks the *status* of the reseed process and returns the *status* and a *Null*
3726 string if the reseed process was not successful.

3727 Step 2.3.3 requests the generation of 256 bits.

3728 Step 2.3.4 checks the *status* of the generate process and returns the *status* and a *Null*
3729 string if the generate process was not successful. The "256" could be omitted since it
3730 is known to be the same as the hard-coded security strength.

3731 Step 2.3.5 assembles the full-entropy bitstring.

3732 Step 2.3.6 counts the number of bits assembled so far.

3733 In step 2.4, the *last_call* value is set to one to indicate that the requested bits were
3734 generated by the RBG3(RS) construction rather than by direct use of the DRBG.

3735 3. The *status* and generated bits from the **RBG3(RS)_Generate** function in step 2 are
3736 returned to the RBG3(RS) construction and forwarded to the application in response to
3737 the generate request in step 1.

3738 **B.6.3. Generation by the Directly Accessible DRBG**

3739 Assuming that the DRBG has been instantiated (see Appendix B.6.1), it can be accessed directly
3740 by a consuming application in the same manner as the RBG2(P) example in Appendix B.4.2 using
3741 the *RBG3_DRBG_state_handle* obtained during instantiation (see Appendix B.6.1).
3742 Pseudorandom bits can be generated directly by the CTR_DRBG as follows:

3743 1. An application requests the generation of pseudorandom bits directly from the DRBG
3744 within the RBG3(RS) construction:

3745 $(status, returned_bits) = \text{DRBG_Generate_request}(RBG3_DRBG_state_handle, n, s)$,
3746 where *RBG3_DRBG_state_handle* was obtained during instantiation (see Appendix
3747 B.6.1), *n* is the requested number of bits to be returned, and *s* is the requested security
3748 strength.

3749 2. Upon receiving the generate request, the RBG3(RS) construction executes a
3750 **DRBG_Generate** function rather than an **RBG3(RS)_Generate** function:

3751 $(status, returned_bits) = \text{DRBG_Generate}(RBG3_DRBG_state_handle, n)$.

3752 The security strength parameter (i.e., 256) is omitted since its value has been hard-coded.

3753 The **DRBG_Generate** function specified in SP 800-90A has been modified to determine
3754 whether a reseed is required before generating the requested output by checking the
3755 value of *last_call*. An extraction³¹ of the **DRBG_Generate** function in SP 800-90A is:

3756 [After other preliminary checks have been performed]

3757 If $((last_call = 1) \text{ OR } (reseed_counter > reseed_interval))$, then

3758 $status = \text{DRBG_Reseed}(RBG3_DRBG_state_handle)$.

3759 If $(status \neq \text{SUCCESS})$, then return $(status, Null)$.

3760 ...

3761 $(returned_bits, new_working_state_values) =$
3762 **Generate_algorithm**(*current_working_state_values*, *requested_number_of_bits*).

3763 *last_call* = 0.

3764 [Closing steps to update the internal state]

³¹ The complete **DRBG_Generate** function is significantly longer.

3765 An additional step has also been included above to indicate that this use of the DRBG is direct
3766 rather than part of the RBG3(RS) construction (i.e., setting *last_call* = 0). This step is used to
3767 indicate that if the next use of the DRBG is also by direct access, a reseed is not required before
3768 generating bits.

3769 **B.6.4. Reseeding a DRBG**

3770 When operating as part of the RBG3(RS) construction, the **DRBG_Reseed** function is invoked
3771 one or more times to produce full-entropy output when the **RBG3(RS)_Generate** function is
3772 invoked by a consuming application (see Sec. 6.5.1.3).

3773 When operating as the directly accessible DRBG, the DRBG is reseeded 1) if explicitly requested
3774 by the consuming application, 2) whenever the previous use of the DRBG was by the
3775 **RBG3(RS)_Generate** function (see Appendix B.6.2), or 3) automatically during a
3776 **DRBG_Generate** call at the end of the seedlife of the RBG2(P) construction (see the
3777 **DRBG_Generate** function specification in SP 800-90A).

3778 1. The reseed function is requested by an application using:

3779 $status = \text{DRBG_Reseed}(\text{RBG3_DRBG_state_handle}),$

3780 where *RBG3_DRBG_state_handle* was obtained during instantiation.

3781 2. The **DRBG_Reseed** function is executed in response to a reseed request by an
3782 application (see step 1) or during the generation process (see Appendices B.6.2 and B.6.3):

3783 $status = \text{DRBG_Reseed}(\text{RBG3_DRBG_state_handle}).$

3784 For this example, the **DRBG_Reseed** function is modified to obtain $s + 64$ bits of entropy
3785 rather than the “normal” s bits of entropy (see method A for step 3.1 in Sec. 6.5.1.2.1).

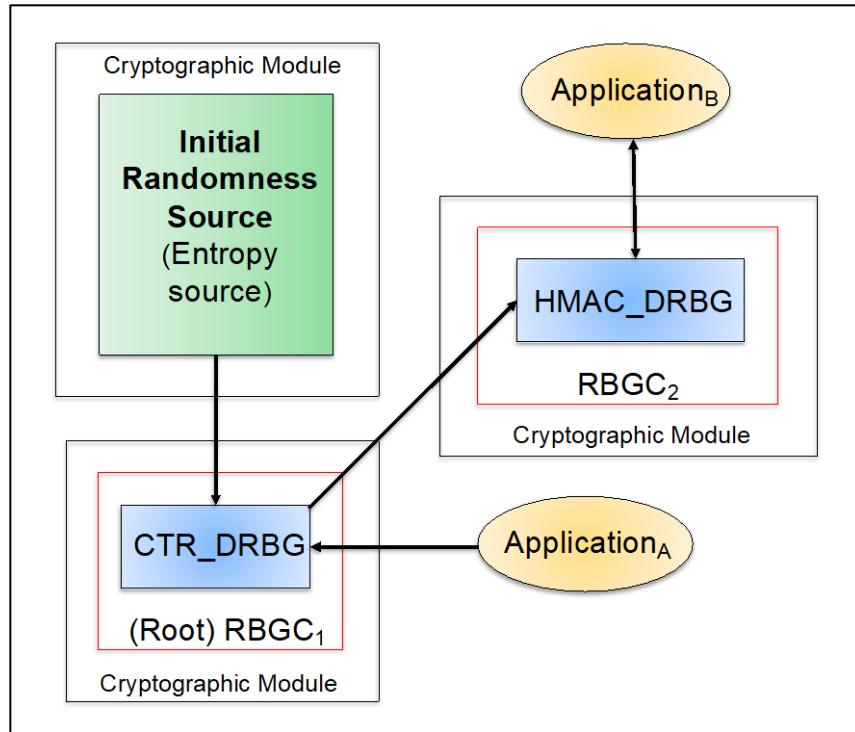
3786 $(status, seed_material) = \text{Get_entropy_bitstring}(s + 64).$

3787 If *status* = SUCCESS is returned by the **DRBG_Reseed** function, the internal state has
3788 been updated with at least 320 bits of fresh entropy (i.e., $256 + 64 = 320$). *Status* =
3789 SUCCESS is returned to the calling application by the **DRBG_Reseed** function.

3790 If *status* ≠ SUCCESS (e.g., the entropy source has failed), the DRBG has not reseeded,
3791 and an error indication is returned as the status from **DRBG_Reseed** function to the
3792 calling application.

3793 **B.7. DRBG Chains Using the RBGC Construction**

3794 A chain of DRBGs consists of RBGC constructions and an initial randomness source on the same
3795 computing platform. For this example, the initial randomness source is a physical entropy source
3796 that provides output with full entropy (i.e., the initial randomness source is a full-entropy source).
3797 The chain includes two RBGC constructions: the root RBGC construction (RBGC_1) and a child
3798 (RBGC_2) (see Fig. 51).



3799

Fig. 51. Example of DRBG chains

3800 In this example, a CTR_DRBG with no derivation function is used in the root (RBGC₁). It will be
3801 seeded and reseeded at a security strength of 192 bits using the initial randomness source.
3802 RBGC₂ is implemented using SHA-256 and the HMAC_DRBG. RBGC₂ will be seeded and
3803 reseeded at a security strength of 128 bits using the root (RBGC₁) as its randomness source.

3805 **B.7.1. Instantiation of the RBGC Constructions**

3806 The DRBG in each RBGC construction is instantiated by an application using a known randomness
3807 source, starting with the instantiation of the DRBG in the root using the initial randomness source
3808 (see Appendix B.7.1.1). Subsequent layers in the chain can be instantiated when an already-
3809 instantiated RBGC construction is available. For this example, after the root has been
3810 instantiated, the DRBG in a child RBGC construction (RBGC₂) can be instantiated using the root
3811 as its randomness source (see Sec. 7.2.1.2).

3812 **B.7.1.1. Instantiation of the Root RBGC Construction**

3813 The root of the DRBG chain is instantiated using the initial randomness source, which for this
3814 example is an entropy source that provides output with full entropy. The instantiation is
3815 requested by an application (i.e., Application_A in Fig. 51). The CTR_DRBG in the root is
3816 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:

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.

- Upon receiving the instantiate request, the root (RBGC₁) executes the instantiate function for its DRBG:

$(status, Root_DRBG_state_handle) = \text{DRBG_Instantiate}(192, \text{"Root RBGC"}).$

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:

$(status, seed\ material) = \text{Get_entropy_bitstring}(320, Method\ I).$

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 (RBC₁) 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 RBGC₂ 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 RBGC₂) 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 RBGC₂.

The DRBG in RBGC₂ 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”),

3853 where “RBGC2 DRBG” is the personalization string, and *RBGC2_DRBG_state_handle* is
3854 the name of the state handle to be assigned to the DRBG in the RBGC₂ construction.

- 3855 2. Upon receiving the instantiate request, the RBGC₂ construction executes the instantiate
3856 function for its DRBG:

3857 $(status, RBGC2_DRBG_state_handle) = \text{DRBG_Instantiate}(128, "RBGC2 DRBG")$.

3858 The **DRBG_Instantiate** function in the DRBG sends a generate request to the root:

3859 $(status, seed_material) = \text{DRBG_Generate}(Root_DRBG_state_handle, 192, 128)$,

3860 where

- 3861 • *Root_DRBG_state_handle* is the state handle for the internal state of the DRBG in
3862 the root (see Sec. 7.1.1).
- 3863 • The requested security strength is 128 bits, so for the HMAC_DRBG in RBGC₂,
3864 the number of bits requested from the root (i.e., RBGC₂’s randomness source) is
3865 $3s/2 = 192$ bits.

3866 See Appendix B.7.2 for the handling of a generate request by an RBGC construction.

3867 If the *status* returned from the randomness source (RBGC₁) in response to the generate
3868 request indicates a success, the HMAC_DRBG in RBGC₂ is seeded using the
3869 *seed_material* returned from the generate request (Appendix B.7.2) and the
3870 *personalization_string* (“RBGC2 DRBG”) from the instantiate request in step 1 (see SP 800-
3871 90A). The internal state is recorded (including the security strength of the instantiation),
3872 and the *status* and the state handle are returned to the RBGC₂ construction to be
3873 forwarded to the application that requested the instantiation of the DRBG in the RBGC₂
3874 construction.

3875 If the *status* indicates an error, then the internal state is not created. The *status* and an
3876 invalid state handle are returned to the RBGC₂ construction to be forwarded to the
3877 application that requested the instantiation of the DRBG in the RBGC₂ construction.

3878 B.7.2. Requesting the Generation of Pseudorandom Bits

- 3879 1. An application or a child RBGC construction requests the generation of pseudorandom
3880 bits as follows:

3881 $(status, seed_material) = \text{DRBG_Generate_request}(DRBG_state_handle, n, s)$,

3882 where

- 3883 • *DRBG_state_handle* is the state handle for the internal state of the DRBG in the
3884 RBGC construction requested to generate the bits. For this example, the state
3885 handle is *Root_DRBG_state_handle* for the DRBG in the root RBGC construction.
3886 For RBGC₂, the state handle is *RBGC2_DRBG_state_handle*.
- 3887 • *n* is the number of bits to be generated using the DRBG in the RBGC construction.

- 3888 • s is the required security strength to be supported by the DRBG in the RBGC
3889 construction.
- 3890 2. Upon receiving the generate request, the RBGC construction executes the generate
3891 function for its DRBG:
- 3892 $(status, seed_material) = \text{DRBG_Generate}(DRBG_state_handle, n, s)$.
- 3893 If the returned $status$ indicates success, the requested number of bits are returned
3894 ($seed_material$) to the RBGC construction and forwarded to the requesting entity with the
3895 $status$. The requesting entity is either an application or a child of the RBGC construction.
- 3896 If the returned status indicates an error, $seed_material$ is a *Null* bitstring. This could, for
3897 example, be the result of requesting a higher security strength than is instantiated for the
3898 DRBG requested to generate bits. The $status$ and the *Null* bitstring are returned to the
3899 RBGC construction and forwarded to the requesting entity.

3900 **B.7.3. Reseeding an RBGC Construction**

3901 The DRBG in an RBGC construction may be explicitly requested to be reseeded by an application,
3902 or the DRBG may automatically reseed itself (e.g., at the end of its seedlife or after some system
3903 interrupt).

- 3904 1. An application requests the reseed of the DRBG in an RBGC construction as follows:

3905 $(status) = \text{DRBG_Reseed_request}(DRBG_state_handle)$.

3906 $DRBG_state_handle$ is $Root_DRBG_state_handle$ for $RBGC_1$ and
3907 $RBG2_DRBG_state_handle$ for $RBGC_2$.

- 3908 2. Upon receiving a reseed request or if scheduled for automatic reseeding, the RBGC
3909 construction executes the reseed function for its DRBG:

3910 $status = \text{DRBG_Reseed}(DRBG_state_handle)$.

3911 Appendix B.7.3.1 discusses the reseed function in the root's DRBG, and Appendix B.7.3.2
3912 discusses the reseed function in the DRBG of $RBGC_2$.

3913 **B.7.3.1. Reseeding the Root RBGC Construction**

3914 The **DRBG_Reseed** function in the root uses the initial randomness source to reseed in the same
3915 manner as for instantiation (i.e., by sending a **Get_entropy_bitstring** request to the entropy
3916 source). For the CTR_DRBG in the root, 320 bits are again requested:

3917 $(status, seed_material) = \text{Get_entropy_bitstring}(320, Method_1)$.

3918 If the returned $status$ indicates a success, $seed_material$ is returned from the initial randomness
3919 source, and the CTR_DRBG within the root is reseeded using the $seed_material$ (see SP 800-
3920 90A). The DRBG's internal state is updated, and the $status$ is returned to the application by the
3921 **DRBG_Reseed** function in the root RBGC construction.

3922 If the *status* indicates an error, then the internal state is not updated. The *status* is returned to
3923 the application.

3924 **B.7.3.2. Reseeding a Child RBGC Construction**

3925 The **DRBG_Reseed** function in the RBGC construction uses its randomness source in the same
3926 manner as for instantiation (i.e., by sending a **DRBG_Generate_request** to its randomness
3927 source, which is the root in this example).

3928 For the HMAC_DRBG in RBGC_2 , $s = 128$ bits are requested from the root RBGC construction
3929 (where s is the security strength of the DRBG instantiation in RBGC_2 ; see Appendix B.7.1.2).

3930 $(\text{status}, \text{seed_material}) = \text{DRBG_Generate}(\text{Root_DRBG_state_handle}, 128, 128),$

3931 where:

- $\text{Root_DRBG_state_handle}$ is the state handle for the internal state of the DRBG in the
3933 root (see Appendix B.7.1.1).
- The requested security strength is 128 bits, so for the HMAC_DRBG in RBGC_2 , the
3935 number of bits requested from the root (RBGC₂'s randomness source) is $s = 128$ bits.

3936 Appendix B.7.2 discusses the handling of a generate request by an RBGC construction.

3937 **Appendix C. Addendum to SP 800-90A: Instantiating and Reseeding a CTR_DRBG**

3938 The derivation functions in this appendix will be included in the next revision of SP 800-90A along
3939 with other changes that are needed for consistency with this version of SP 800-90C.

3940 **C.1. Background and Scope**

3941 The CTR_DRBG, specified in SP 800-90A, uses the AES block cipher in FIPS 197 and has two
3942 versions that may be implemented: with or without a derivation function.

3943 When a derivation function is not used, SP 800-90A requires the use of seed material with full
3944 entropy for instantiating and reseeding a CTR_DRBG. This addendum permits the use of an RBG
3945 compliant with SP 800-90C to provide the required seed material for the CTR_DRBG when
3946 implemented as specified in SP 800-90C (see Appendix C.2).

3947 When a derivation function is used in a CTR_DRBG implementation, SP 800-90A specifies the
3948 use of the block cipher derivation function. This addendum modifies the requirements in SP 800-
3949 90A for the CTR_DRBG by specifying two additional derivation functions that may be used
3950 instead of the block cipher derivation function (see Appendix C.3).

3951 **C.2. CTR_DRBG Without a Derivation Function**

3952 When a derivation function is not used, SP 800-90A requires that *seedlen* full-entropy bits be
3953 provided as the seed material (e.g., from an entropy source that provides full-entropy output),
3954 where *seedlen* is the length of the key to be used by the CTR_DRBG plus the length of the output
3955 block (i.e., 128 bits for AES). SP 800-90C includes an approved method for externally conditioning
3956 the output of an entropy source to provide a bitstring with full entropy when using an entropy
3957 source that does not provide full-entropy output.

3958 SP 800-90C also permits the use of seed material from an RBG when the DRBG to be instantiated
3959 and reseeded is implemented and used as specified in SP 800-90C.

3960 **C.3. CTR_DRBG Using a Derivation Function**

3961 When a derivation function is used within a CTR_DRBG, SP 800-90A specifies the use of the
3962 **Block_cipher_df** included in that document during instantiation and reseeding to adjust the
3963 length of the seed material to *seedlen* bits, where

3964
$$\text{seedlen} = \text{the security strength} + \text{the block length.}$$

3965 For AES, *seedlen* = 256, 320, or 384 bits (see SP 800-90A). During generation, the length of any
3966 additional input provided during the generation request is also adjusted to *seedlen* bits.

3967 Two alternative derivation functions are specified in Appendices C.3.2 and C.3.3. Appendix C.3.1
3968 discusses the keys and constants for use with the alternative derivation functions specified in
3969 Appendices C.3.2 and C.3.3.

3970 **C.3.1. Derivation Keys and Constants**

3971 Both of the derivation methods specified in Appendices C.3.2 and C.3.3 use an AES derivation key
3972 (*df_Key*) whose length **shall** meet or exceed the instantiated security strength of the DRBG
3973 instantiation. The *df_Key* **may** be set to any value and **may** be the current value of a key used by
3974 the DRBG.

3975 These alternative methods use three 128-bit constants C_1 , C_2 , and C_3 , which are defined as:

3976 $C_1 = 000000\dots00$

3977 $C_2 = 101010\dots10$

3978 $C_3 = 010101\dots01$

3979 The value of B used in Appendices C.3.2 and C.3.3 depends on the length of the AES derivation
3980 key (*df_Key*). When the length of *df_Key* = 128 bits, then $B = 2$. Otherwise, $B = 3$.

3981 **C.3.2. Derivation Function Using CMAC**

3982 CMAC is a block-cipher mode of operation specified in SP 800-38B. The CMAC_df derivation
3983 function is specified as follows:

3984 **CMAC_df:**

3985 **Input:** bitstring *input_string*, integer *number_of_bits_to_return*.

3986 **Output:** bitstring *Z*.

3987 **Process:**

3988 1. Let C_1 , C_2 , and C_3 be 128-bit blocks defined as 000000...0, 101010...10, and 010101...01,
3989 respectively.

3990 2. Get *df_Key*. Comment: See Appendix C.3.1.

3991 3. *Z* = the Null string.

3992 4. For $i = 1$ to B :

3993 $Z = Z \parallel \text{CMAC}(df_Key, C_i \parallel input_string)$.

3994 5. *Z* = **leftmost**(*Z*, *number_of_bits_to_return*).

3995 6. Return(*Z*).

3996 **C.3.3. Derivation Function Using CBC-MAC**

3997 This CBC-MAC derivation function **shall** only be used when the *input_string* has the following
3998 properties:

- 3999 • The length of the *input_string* is always a fixed length.

- 4000 • The length of the *input_string* is an integer multiple of 128 bits. Let m be the number of
4001 128-bit blocks in the *input_string*.

4002 This derivation function is specified as follows:

4003 CBC-MAC df:

4004 **Input:** bitstring *input_string*, integer *number_of_bits_to_return*.

4005 **Output:** bitstring Z .

4006 Process:

- 4009 2. Get *df Key*. Comment: See Appendix C.3.1.

4010 3. Z = the *Null* string.

- 4011 4. Let $\text{input_string} = S_1 \parallel S_2 \parallel \dots \parallel S_m$, where the S_i are contiguous 128-bit blocks.

4012 5. For $j = 1$ to B :

- $$4013 \qquad 5.1 \qquad S_0 = C_j.$$

4014 5.2 $V = 128$ -bit blo

4015 5.3 For $i = 0$ to m :
 4016 $V = \text{Encrypt}(df_Key, V \oplus S_i)$. Comment: Perform the cipher
 4017 operation specified in FIPS 197

⁴⁰¹⁸ 5.4. $Z = Z \parallel V$

⁴⁰¹⁹ 6 $Z \equiv \text{leftmost}(Z, n)$

4020 7 Return(Z)

4021

4022 **Appendix D. List of Abbreviations and Acronyms**

4023 **AES**

4024 Advanced Encryption Standard³²

4025 **CAVP**

4026 Cryptographic Algorithm Validation Program

4027 **CMVP**

4028 Cryptographic Module Validation Program

4029 **DRBG**

4030 Deterministic Random Bit Generator³³

4031 **FIPS**

4032 Federal Information Processing Standard

4033 **MAC**

4034 Message Authentication Code

4035 **NIST**

4036 National Institute of Standards and Technology

4037 **RBG**

4038 Random Bit Generator

4039 **SP**

4040 (NIST) Special Publication

4041 **Sub-DRBG**

4042 Subordinate DRBG

4043 **TDEA**

4044 Triple Data Encryption Algorithm³⁴

4045 **XOR**

4046 Exclusive-Or (operation)

4047 **D.1. List of Symbols**

4048 **0^x**

4049 A string of x zeroes.

4050 **⌈x⌉**

4051 The ceiling of x ; the least integer number that is not less than the real number x . For example, $\lceil 3 \rceil = 3$, and $\lceil 5.5 \rceil = 6$.

4053 **ε**

4054 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.

4055	min(<i>a</i>, <i>b</i>)
4056	The minimum of <i>a</i> and <i>b</i> .
4057	output_len
4058	The bit length of the output block of a cryptographic primitive.
4059	s
4060	The security strength.
4061	X ⊕ Y
4062	Boolean bitwise exclusive-or (also bitwise addition modulo 2) of two bitstrings <i>X</i> and <i>Y</i> of the same length.
4063	+
4064	Addition over real numbers.
4065	X Y
4066	The concatenation of two bitstrings <i>X</i> and <i>Y</i> .
4067	

4068 **Appendix E. Glossary**

4069 **additional input**

4070 Optional additional information that could be provided in a generate or reseed request by a consuming application.

4071 **adversary**

4072 A malicious entity whose goal is to determine, guess, or influence the output of an RBG.

4073 **alternative randomness source**

4074 A sibling of the parent randomness that may be used by a non-root RBGC construction for reseeding when the parent
4075 randomness source is unavailable.

4076 **approved**

4077 An algorithm or technique for a specific cryptographic use that is specified in a FIPS or NIST recommendation,
4078 adopted in a FIPS or NIST recommendation, or specified in a list of NIST-approved security functions.

4079 **backtracking resistance**

4080 A property of a DRBG that provides assurance that compromising the current internal state of the DRBG does not
4081 weaken previously generated outputs. See SP 800-90A for a more complete discussion. Contrast with *prediction*
4082 *resistance*.

4083 **biased**

4084 A random variable is said to be biased if values of the finite sample space are selected with unequal probability.
4085 Contrast with *unbiased*.

4086 **big-endian format**

4087 A format in which the most significant bytes (the bytes containing the high-order or leftmost bits) are stored in the
4088 lowest address with the following bytes in sequentially higher addresses.

4089 **bitstring**

4090 An ordered sequence (string) of 0s and 1s. The leftmost bit is the most significant bit.

4091 **block cipher**

4092 A parameterized family of permutations on bitstrings of a fixed length; the parameter that determines the
4093 permutation is a bitstring called the key.

4094 **computing platform**

4095 A system's hardware, firmware, operating system, and all applications and libraries executed by that operating
4096 system. Components that communicate with the operating system through a peripheral bus or a network, either
4097 physical or virtual, are not considered to be part of the same computing platform.

4098 **conditioning function (external)**

4099 As used in SP 800-90C, a deterministic function that is used to produce a bitstring with full entropy or to distribute
4100 entropy.

4101 **consuming application**

4102 An application that uses random outputs from an RBG.

4103 **cryptographic boundary**

4104 An explicitly defined physical or conceptual perimeter that establishes the physical and/or logical bounds of a
4105 cryptographic module and contains all the hardware, software, and/or firmware components of a cryptographic
4106 module.

- 4107 **cryptographic module**
4108 The set of hardware, software, and/or firmware that implements cryptographic functions (including cryptographic
4109 algorithms and key generation) and is contained within the cryptographic boundary.
- 4110 **deterministic random bit generator (DRBG)**
4111 An RBG that produces random bitstrings by applying a deterministic algorithm to seed material.
4112 *Note:* A DRBG at least has access to a randomness source initially.
- 4113 **digitization**
4114 The process of generating raw discrete digital values from non-deterministic events (e.g., analog noise sources)
4115 within a noise source.
- 4116 **DRBG chain**
4117 A chain of DRBGs in which one DRBG is used to provide seed material for another DRBG.
- 4118 **entropy**
4119 A measure of disorder, randomness, or variability in a closed system.
4120 *Note1:* The entropy of a random variable X is a mathematical measure of the amount of information gained
4121 by an observation of X .
4122 *Note2:* The most common concepts are Shannon entropy and min-entropy. Min-entropy is the measure
4123 used in SP 800-90.
- 4124 **entropy rate**
4125 The validated rate at which an entropy source provides entropy in terms of bits per entropy-source output (e.g., five
4126 bits of entropy per 8-bit output sample).
- 4127 **entropy source**
4128 The combination of a noise source, health tests, and an optional conditioning component that produce bitstrings
4129 containing entropy. A distinction is made between entropy sources with physical noise sources and those having
4130 non-physical noise sources.
- 4131 **fresh entropy**
4132 A bitstring that is output from a non-deterministic randomness source that has not been previously used to generate
4133 output or has not otherwise been made externally available.
4134 *Note:* The randomness source should be an entropy source or RBG3 construction.
- 4135 **fresh randomness**
4136 A bitstring that is output from a randomness source that has not been previously used to generate output or has not
4137 otherwise been made externally available.
- 4138 **full-entropy bitstring**
4139 A bitstring with ideal randomness (i.e., the amount of entropy per bit is equal to 1). This recommendation assumes
4140 that a bitstring has *full entropy* if the entropy rate is at least $1 - \varepsilon$, where ε is at most 2^{-32} .
- 4141 **full-entropy source**
4142 An SP 800-90B-compliant entropy source that has been validated as providing output with full entropy or the
4143 validated combination of an SP 800-90B-compliant entropy source and an external conditioning function that
4144 provides full-entropy output.

4145 **hash function**

4146 A (mathematical) function that maps values from a large (possibly very large) domain into a smaller range. The
4147 function satisfies the following properties:

4148 1. (One-way) It is computationally infeasible to find any input that maps to any pre-specified output.

4149 2. (Collision-free) It is computationally infeasible to find any two distinct inputs that map to the same output.

4150 **health testing**

4151 Testing within an implementation immediately prior to or during normal operation to obtain assurance that the
4152 implementation continues to perform as implemented and validated.

4153 *Note:* Health tests are comprised of continuous tests and startup tests.

4154 **ideal randomness source**

4155 The source of an ideal random sequence of bits. Each bit of an ideal random sequence is unpredictable and unbiased
4156 with a value that is independent of the values of the other bits in the sequence. Prior to an observation of the
4157 sequence, the value of each bit is equally likely to be 0 or 1, and the probability that a particular bit will have a
4158 particular value is unaffected by knowledge of the values of any or all the other bits. An ideal random sequence of n
4159 bits contains n bits of entropy.

4160 **independent entropy sources**

4161 Two entropy sources are *independent* if knowledge of the output of one entropy source provides no information
4162 about the output of the other entropy source.

4163 **initial randomness source**

4164 The randomness source for the root RBGC construction in a DRBG chain of RBGC constructions.

4165 **instantiate**

4166 The process of initializing a DRBG with sufficient randomness to generate pseudorandom bits at the desired security
4167 strength.

4168 **internal state (of a DRBG)**

4169 The collection of all secret and non-secret information about an RBG or entropy source that is stored in memory at
4170 a given point in time.

4171 **known answer test**

4172 A test that uses a fixed input/output pair to detect whether a deterministic component was implemented correctly
4173 or continues to operate correctly.

4174 **min-entropy**

4175 A lower bound on the entropy of a random variable. The precise formulation for min-entropy is $(-\log_2 \max p_i)$ for a
4176 discrete distribution having probabilities p_1, \dots, p_k . Min-entropy is often used as a measure of the unpredictability of
4177 a random variable.

4178 **must**

4179 Used to indicate a requirement that may not be testable by a CMVP testing lab.

4180 *Note:* **Must** may be coupled with **not** to become **must not**.

4181 **noise source**

4182 A source of unpredictable data that outputs raw discrete digital values. The digitization mechanism is considered
4183 part of the noise source. A distinction is made between physical noise sources and non-physical noise sources.

4184 **non-physical entropy source**

4185 An entropy source whose primary noise source is non-physical.

- 4186 **non-physical noise source**
4187 A noise source that typically exploits system data and/or user interaction to produce digitized random data.
- 4188 **non-validated entropy source**
4189 An entropy source that has not been validated by the CMVP as conforming to SP 800-90B.
- 4190 **null string**
4191 An empty bitstring.
- 4192 **parent randomness source**
4193 The randomness source used to seed a non-root RBGC construction.
- 4194 **personalization string**
4195 An optional input value to a DRBG during instantiation.
- 4196 **physical entropy source**
4197 An entropy source whose primary noise source is physical.
- 4198 **physical noise source**
4199 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.
- 4202 **physically secure channel**
4203 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.
- 4207 **prediction resistance**
4208 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.
4211 (Contrast with *backtracking resistance*.)
- 4212 For an RBG, compromising the output of the RBG does not allow future outputs of the RBG to be predicted.
- 4213 **pseudocode**
4214 An informal, high-level description of a computer program, algorithm, or function that resembles a simplified programming language.
- 4216 **random bit generator (RBG)**
4217 A device or algorithm that outputs a random sequence that is effectively indistinguishable from statistically independent and unbiased bits.
- 4219 **randomness**
4220 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.
- 4224 **randomness source**
4225 A source of randomness for an RBG. The randomness source may be an entropy source or an RBG construction.

- 4226 **RBG1 construction**
4227 An RBG construction with the DRBG and the randomness source in separate cryptographic modules.
- 4228 **RBG2 construction**
4229 An RBG construction with one or more entropy sources and a DRBG within the same cryptographic module. This RBG
4230 construction does not provide full-entropy output.
- 4231 Note: An RBG2 construction may be either an RBG2(P) or RBG2(NP) construction.
- 4232 **RBG2(NP) construction**
4233 A non-physical RBG2 construction that obtains entropy from one or more validated non-physical entropy sources
4234 and possibly from one or more validated physical entropy sources. This RBG construction does not provide full-
4235 entropy output.
- 4236 **RBG2(P) construction**
4237 A physical RBG2 construction that includes a DRBG and one or more entropy sources in the same cryptographic
4238 module. Only the entropy from validated physical entropy sources is counted when fulfilling an entropy request
4239 within the RBG. This RBG construction does not provide full-entropy output.
- 4240 **RBG3 construction**
4241 An RBG construction that includes a DRBG and one or more entropy sources in the same cryptographic module.
4242 When working properly, bitstrings that have full entropy are produced. Sometimes called a *non-deterministic*
4243 *random bit generator* (NRBG) or true random number (or bit) *generator*.
- 4244 **RBGC construction**
4245 An RBG construction used within a DRBG chain in which one DRBG is used to provide seed material for another
4246 DRBG. The construction does not provide full-entropy output.
- 4247 **reseed**
4248 To refresh the internal state of a DRBG with seed material from a randomness source.
- 4249 **root RBGC construction**
4250 The first RBGC construction in a DRBG chain of RBGC constructions.
- 4251 **sample space**
4252 The set of all possible outcomes of an experiment.
- 4253 **security boundary**
4254 For an entropy source, a conceptual boundary that is used to assess the amount of entropy provided by the values
4255 output from the entropy source. The entropy assessment is performed under the assumption that any observer
4256 (including any adversary) is outside of that boundary during normal operation.
- 4257 For a DRBG, a conceptual boundary that contains the required DRBG functions and the DRBG's internal state.
- 4258 For an RBG, a conceptual boundary that is defined with respect to one or more threat models that includes an
4259 assessment of the applicability of an attack and the potential harm caused by the attack.
- 4260 **security strength**
4261 A number associated with the amount of work (i.e., the number of basic operations of some sort) that is required to
4262 "break" a cryptographic algorithm or system in some way. In this recommendation, the security strength is specified
4263 in bits and is a specific value from the set {128, 192, 256}. If the security strength associated with an algorithm or
4264 system is s bits, then it is expected that (roughly) 2^s basic operations are required to break it.
- 4265 Note: This is a classical definition that does not consider quantum attacks. This definition will be revised to
4266 address quantum issues in the future.

- 4267 **seed**
4268 Verb: To initialize or update the internal state of a DRBG with seed material and (optionally) a personalization string
4269 or additional input. The seed material should contain sufficient randomness to meet security requirements.
- 4270 Noun: The combination of seed material and (optional) personalization or additional input.
- 4271 **seed material**
4272 An input bitstring from a randomness source that provides an assessed minimum amount of randomness (e.g.,
4273 entropy) for a DRBG.
- 4274 **seedlife**
4275 The period of time between instantiating or reseeding a DRBG with seed material and either reseeding the DRBG
4276 with seed material containing new, unused randomness or uninstantiating the DRBG.
- 4277 **shall**
4278 The term used to indicate a requirement that is testable by a testing lab. See *testable requirement*.
4279 *Note: Shall* may be coupled with **not** to become **shall not**.
- 4280 **should**
4281 The term used to indicate an important recommendation. Ignoring the recommendation could result in undesirable
4282 results.
4283 *Note: Should* may be coupled with **not** to become **should not**.
- 4284 **sibling (randomness source)**
4285 A sibling of the parent randomness source for a non-root RBGC construction (i.e., the sibling can be considered the
4286 “aunt” or “uncle” in “human family” terms). The “grandparent” of the non-root RBGC construction is the parent of
4287 both the parent randomness source and the sibling.
- 4288 **state handle**
4289 A pointer to the internal state information for a particular DRBG instantiation.
- 4290 **subordinate DRBG (sub-DRBG)**
4291 A DRBG that is instantiated by an RBG1 construction and contained within the same security boundary as the RBG1
4292 construction.
- 4293 **support a security strength (by a DRBG)**
4294 The DRBG has been instantiated at a security strength that is equal to or greater than the security strength requested
4295 for the generation of random bits.
- 4296 **targeted security strength**
4297 The security strength that is intended to be supported by one or more implementation-related choices (e.g.,
4298 algorithms, cryptographic primitives, auxiliary functions, parameter sizes, and/or actual parameters).
- 4299 **testable requirement**
4300 A requirement that can be tested for compliance by a testing lab via operational testing, code review, or a review of
4301 relevant documentation provided for validation. A testable requirement is indicated using a **shall** statement.
- 4302 **threat model**
4303 A description of a set of security aspects that need to be considered. A threat model can be defined by listing a set
4304 of possible attacks along with the probability of success and the potential harm from each attack.
- 4305 **unbiased**
4306 A random variable is said to be unbiased if all values of the finite sample space are chosen with the same probability.
4307 Contrast with *biased*.

4308 **uninstantiate**

4309 The termination of a DRBG instantiation.

4310 **validated entropy source**

4311 An entropy source that has been successfully validated by the CAVP and CMVP for conformance to SP 800-90B.