Part 4: Constructions for Building and Validating RBGs

0 New Definitions for Glossary[1]

a. *RBG* -- the full design that produces random bits, including any entropy sources, determinsitic algorithms, reseeding rules, buffering~~, etc.~~, used.

b. *DRBG* -- An RBG providing only computational security, meaning that the RBG has a specific maximum security level (i.e., one of {112,128,192,256}), depending on the amount of entopy provided during instantiation, and that there is some amount of work no less than the $2^{security\ level}$ operations such that an attacker can distinguish the output sequence from the DRBG from an ideal random output sequence. A DRBG may support *prediction resistance*, as described in Part 3.

c. *NRBG* -- An RBG providing information-theoretic security, meaning that there is no amount of work for which an attacker can expect to be able to distinguish the output sequence from an ideal random output sequence.

d. *Basic NRBG*--An NRBG that relies only upon the underlying entropy source. If the entropy source fails to behave according to its assessed entropy, there is no guarantee of security made for the sequence of output bits from the NRBG.

e. *Enhanced NRBG*--An NRBG that provides a guaranteed fallback to an approved DRBG, in case the entropy source fails to behave according to its assessed entropy.

f. *Composite RBG*-- An RBG capable of supporting requests for both computationally-secure random bits and information-theoretically secure random bits~~, usually with some large performance difference~~.

g. *critical failure* -- A failure of an entropy source that leads to a major loss of security in the surrounding RBG. For DRBGs and constructions that guarantee a fallback to DRBGs, a critical failure happens when the DRBG is insecurely instantiated--somewhat arbitrarily defined for purposes of this standard as being instantiated with 16 or more fewer bits of min-entropy than the DRBG's claimed security level. For Basic NRBGs, a critical failure happens when any noticeable statistical flaw occurs in their output sequence, or when any k bit output has less than $0.95\,k$ bits of min-entropy.[2]

h. *entropy accumulation* -- The process of gradually accumulating the entropy from a long sequence of entropy source outputs, without storing the full output sequence. In Part 4, all accumulation discussed is happening *outside* the entropy source, with no knowledge

Comment [ebb1]: Page: 1
Alphabetize.

Comment [ebb2]: Page: 1
There is too much information here for a definition.

Comment [ebb3]: Page: 1
Need to determine the "standard"/common definition.

Comment [ebb4]: Page: 1
Too much information for a definition.

Comment [ebb5]: Page: 1
This doesn't seem to be enough; need to mention a DRBG?

Comment [ebb6]: Page: 1
This text is too much for the definition; put within the document?

Comment [ebb7]: Page: 1
What does this mean? Is it needed in the definition?

1This whole section needs more entries, but I haven't had time yet.

2My attempt at talking about computational security bounds here won't work, because we would need to count on the validation labs/process determining how much computational security was in someone's nonstandard algorithm, and that's not a reasonable thing to demand of the validation labs. [Do we need to address an non-standard algorithm?]

about the nature of the source except what appears in its entropy estimates.

Comment [ebb8]: Page: 2
Doesn't belong in the definition.

i. *entropy buffering* -- The process of storing accumulated entropy from the entropy source, to allow the handling of occasional bursts of ~~requested~~ entropy from a relatively slow entropy source. In Part 4, all buffering discussed is happening *outside* the entropy source.

Comment [ebb9]: Page: 2
Doesn't belong in the definition.

j. *external conditioning* -- The process of mapping a regular entropy source's outputs to full-entropy outputs from *outside* the entropy source boundary, thus without any detailed information about the entropy source's behavior except for its entropy estimates.

Comment [ebb10]: Page: 2
Shouldn't be in the definition - discuss elsewhere.

k. *Conditioned entropy source*--An entropy source whose output bits are assessed at full entropy.

Comment [ebb11]: Page: 2
Probably need to indicate that the conditioning is done within the entropy source.

l. *External conditioning*--Conditioning of an entropy source's outputs, done in a generic way, outside the entropy source itself, and thus without any assumptions about the model of the underlying entropy source except the assumption that the assessed min-entropy is correct.

Comment [ebb12]: Page: 2
A second definition of external conditioning.

m. *Persistent state*--memory or state for an RBG that is not lost on power-down.

1 Introduction

The preceding parts of this document have:

a. Provided definitions of fundamental concepts, such as entropy, randomness, and security levels, and framed the problem of random bit generation for cryptographic and security applications,

b. Provided guidance for developing approved *entropy sources*, mechanisms that provide truly unpredictable values from some nondeterministic process, and

c. Specified a number of *DRBG* mechanisms containing cryptographic algorithms that, used correctly, are expected to produce bits indistinguishable from ideal random bits up to the specified security level of the instantiation.

Part Four describes how the components and concepts from the previous three parts of this Standard are to be combined into working RBGs--systems for using some source of ultimate unpredictability to produce output bits that are sufficiently close to ideal random bits for some specified purpose as specified by a (possibly infinite) security level.

The remainder of the document consists of discussions of tasks that must be accomplished to design, implement, and validate an approved RBG, and *constructions* for accomplishing some of these tasks.

A construction is a specified way of doing something, such as externally accumulating entropy from an entropy source. Where one or more constructions are given for some task, they represent the only acceptable ways of doing that task within an ANSI X9.82 approved RBG. In this document, constructions are explicitly specified by the heading **"Construction:"**.

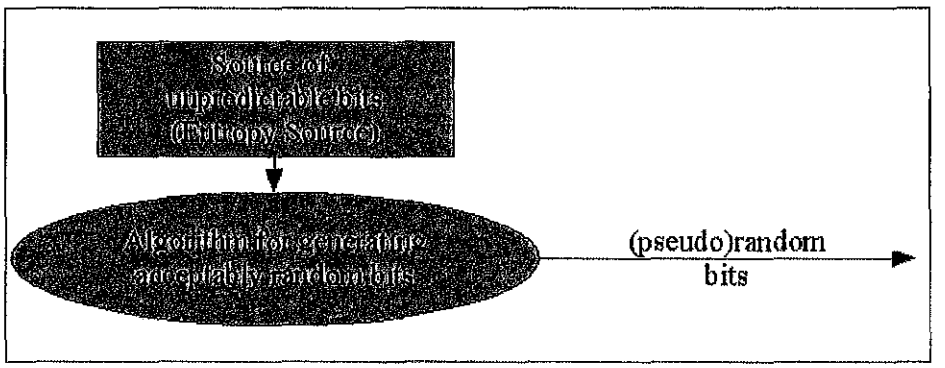
An RBG design is all the parts of an RBG that are not implementation specific.

Comment [ebb13]: Page: 3
Don't understand this.

In the remainder of this document, when an entropy source, NRBG, DRBG, Composite RBG, or RBG is discussed, it should be assumed to refer to an ANSI X9.82-approved entropy source, NRBG, DRBG, etc., unless stated otherwise.

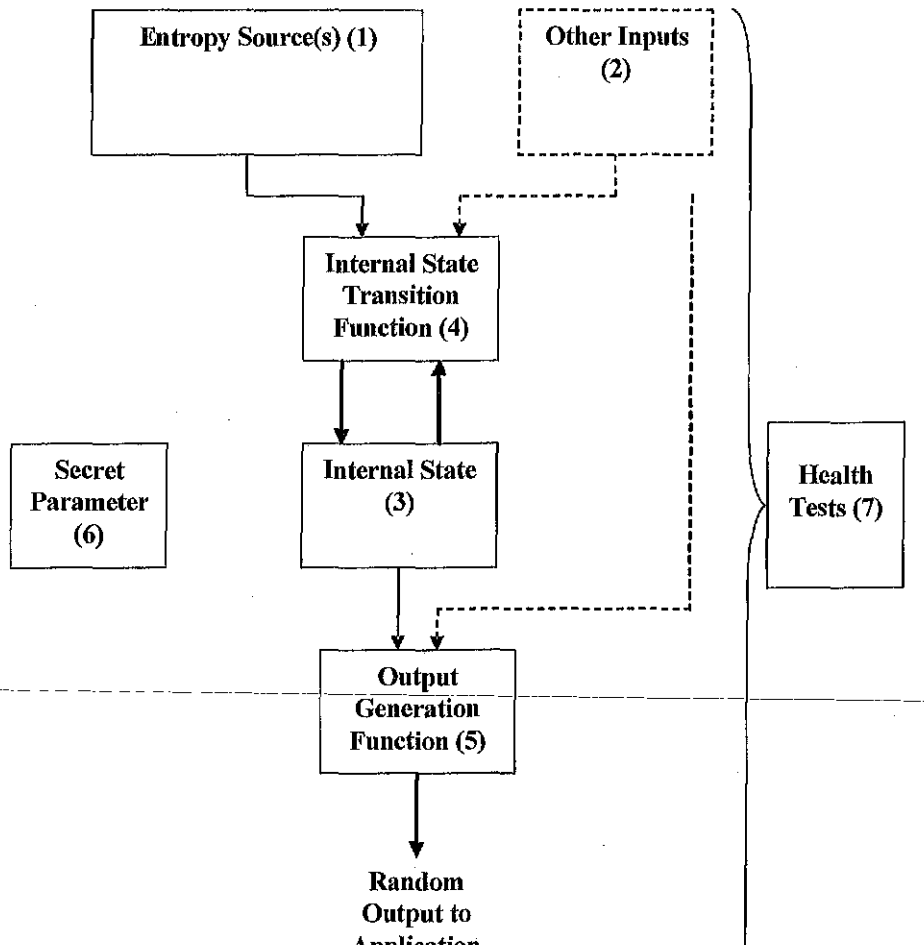
1.1 Structure of an RBG

An RBG produces random bits for some consuming application, providing some assurances about the difficulty of distinguishing its output sequence from an ideal random sequence (that is, a sequence of unbiased, independent, identically distributed bits). Any RBG must consist of some ultimate source of unpredicability (an entropy source) to provide an unguessable state ~~of some kind~~, and some deterministic algorithm to generate random bits from that unguessable state (typically a DRBG algorithm). The basic problem in building a working RBG of any kind is in managing the entropy in the system, and in using the deterministic components in ways that do not violate their security requirements.



Part One of this standard describes a much more detailed model, based on a state-machine view of the process. Part Two ~~of this standard~~ focuses upon the Entropy Sources ~~in this model~~; Part Three ~~of this standard~~ focuses upon choices of the State Transition and Output functions that provide strong security assurances under a variety of allowable ways of interacting with the entropy source. Part Four describes the whole model, focusing on how the components may be combined to achieve a desired level of security.

Comment [ebb14]: Page: 3
Refer to the figure above somewhere.



The model is applied to constructions as follows:[3]...

Comment [ebb15]: Page: 4
I don't think we need to refer to the detailed model in this part; the previous figure or some variation of it is more useful for this Part.

1.2 Entropy Management: How Entropy Sources Go Wrong

The biggest problem in constructing an RBG is managing entropy. There are many

~~3How much time should I give this model? I find this model so general that it's hard to use it directly to describe much in any detail. Unless there's some compelling reason to spend more time on it, I'm going to ignore it from now on~~

reasons for this: algorithms are largely platform- and implementation-independent, but entropy sources are *always* dependent on implementation details--a change in manufacturing processes can convert an excellent entropy source into a terrible one! The characterization of entropy sources is ultimately a matter of experiment and matching an *a priori* sensible model to its observed behavior, and this is a messy and imprecise process. Entropy sources tend to be much less reliable than deterministic components, and testing their behavior in the field tends to be much more difficult. These issues are discussed in much more depth, below.

An important concept to keep in mind is that of a *critical failure of the entropy source*. A critical failure occurs when the entropy source deviates from its expected behavior in a way that is not detected by the RBG, and that causes a real-world security failure. A great deal of the practical work in building a secure RBG amounts to minimizing the probability of a critical failure by providing some level of fallback security in the RBG design, redundancy in components that might fail in some way, and tests to detect the most likely and most damaging kinds of failure.

Comment [ebb16]: Page: 5
Refer to the specific section(s).

Comment [ebb17]: Page: 5
Deviating from expected behavior is a critical failure; it's just more devastating if it's not caught, especially if there is no backup strategy.

1.3 Narrow Pipes, Security Levels, and Cryptanalysis: How Deterministic Algorithms Go Wrong

An RBG either 1) claims some security level from the list of {112,128,192,256}, possibly including a claim of providing prediction resistance, or 2) claims to be information-theoretically secure. For most RBGs, the outputs will be produced directly or indirectly by an approved DRBG ~~algorithm~~. (The exceptions to this are called *Basic NRBGs*, and are discussed below.) The cryptographic strength and other properties of the DRBG ~~algorithms~~ used determine the properties of the RBGs, as is discussed below.

For DRBGs, which claim computational security, the only requirement on the ~~underlying~~ DRBG ~~algorithm~~ is that it be able to support the security level claimed by the DRBG. For NRBGs, which claim information-theoretic security, and for composite mechanisms which sometimes claim information-theoretic security, it is often surprisingly difficult to get more than the design strength of security from the DRBG ~~algorithm~~. For this reason, constructions are provided to accomplish these goals in secure ways.

Comment [ebb18]: Page: 5
A discusion of computational security and information-theoretic security needs to have been discussed before this, either in Part 1 or in this Part.

Comment [ebb19]: Page: 5
Refer to the section(s).

Comment [ebb20]: Page: 5
provide a specific reference.

1.4 Relationships between Entropy Rate and Output Rate

For DRBGs~~,~~ or composite RBGs that ~~can~~ support computationally-secure requests for random bits, there are two requirements on the rate of entropy input: In order to support a given security level s, at least s+64 bits of entropy must be provided to the RBG before any outputs are generated, and at least s new bits of entropy must be provided before any output is generated with prediction resistance. Many DRBG designs may request some multiple of these minimum values before generating outputs, especially before generating the first output, to resist problems with the entropy source.

NRBGs supplied by raw entropy sources require at least $2k$ bits of entropy input for each k-bit output. NRBGs supplied by conditioned entropy sources may use between k and $2k$

Comment [ebb21]: Page: 6
NRBGs always use raw entropy sources. The issue here, I think is whether the entropy source boundary contains the conditioning function.

Comment [ebb22]: Page: 6
Not defined. Will it be defined in Part 2?

bits of entropy input for each *k*-bit output, depending on the details of the NRBG construction used.

~~1.4.1 Rates Supported by Different Entropy Sources~~

Comment [ebb23]: Page: 6
There is a Section 1.4.1 without a Section 1.4.2.

Real-world entropy sources tend to fall into several different categories:

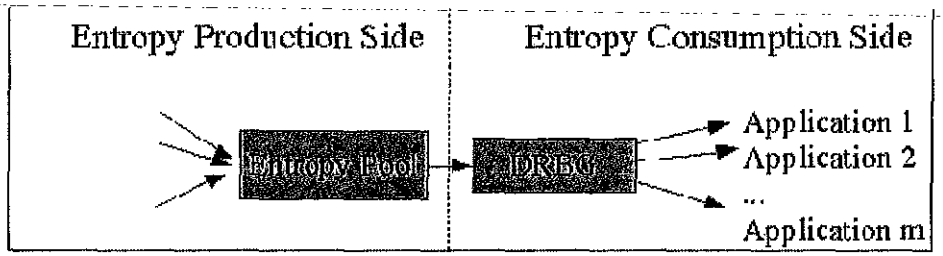
a. Those based on human actions, such as coin flips and precise timing of mouse events. These tend to be relatively slow, and to produce entropy at very different rates at different times.

b. Those based on internal hardware and software events inside a general purpose computer. These tend to produce entropy at a moderate rate under normal conditions. Their rate tends to be variable, based on system loading, network usage, and other conditions, but they are much less variable than sources based on human actions.

c. Those based on noise inside electronic systems. These tend to produce entropy at a high rate, and to be very consistent in their rate of production, given only reasonable operating conditions.

These varying rates of entropy production, especially in internal computer event sources, make buffering (discussed below) an important system design consideration. Software entropy sources **should** be combined ~~together~~ when the availability of certain devices or other resources on a specific platform is unknown.

Comment [ebb24]: Page: 6
The paragraph below seems to indicate that there should be a statement about inconsistent performance here.



Hardware sources, by contrast, tend to have pretty consistent performance over time, ~~and radically different hardware sources are rarely combined~~ together. The process that produces entropy is ongoing, and unless the entropy source is turned off to cut power consumption, its rate of production of entropy can be expected to remain roughly constant. Buffering entropy outputs is useful if a large number of entropy requests are made within a short period of time, or as a way to protect the system from occasional "drop-outs" of the entropy source. Some minimal amount of buffering also masks the timing variations caused by conditioning techniques on the bits produced by the noise source that discard some bits, such as Von Neumann unbiasing.

Comment [ebb25]: Page: 6
Refer to the figure in the text.

Comment [ebb26]: Page: 6
We need not discuss past/present designs here; just what needs to be done.

1.5 A Roadmap

Comment [ebb27]: Page: 7
Needs to be revised in accordance with any changes?

Part Four is arranged as follows: First, the issues raised in implementing, using, and validating an entropy source as part of an RBG are discussed. For reasons that will become clear below, an entropy source must be validated as part of an RBG. Next, constructions for RBGs that support only computationally-bounded security (DRBGs) are provided, along with requirements that must be met outside these constructions. Then, constructions for RBGs providing information theoretic security (NRBGs) and providing both DRBG and NRBG functionality (Composite RBGs) are discussed. Finally, a number of other generally useful constructions are provided. An annex at the end of this document discusses a number of security issues in more depth than is addressed in the main text.

2 Entropy Sources in RBGs[4]

Part 2 describes entropy sources by themselves. The focus in Part 4 is on taking an existing, good entropy source design, and fitting it into the RBG. This includes:

a. Supporting constraints ~~put~~ on the size and rate of the entropy source's outputs by other parts of the RBG's design, by accumulation and buffering outside the entropy source's boundary,

b. Combining entropy source outputs ~~together~~ in a "pool" of bits, and estimating the min-entropy, based on conservative estimation strategies, and

c. Validating the entropy sources for use in the RBG, based on the idea of keeping the probability of a *critical failure* of the entropy source acceptably low. (A critical failure is a failure that leads to a practical security flaw[5].)

2.1 Preliminaries

An *entropy source* is the component of an RBG that provides nondeterministic, unpredictable behavior. An entropy source provides bitstrings containing some entropy as output, and an assessment of the min-entropy of these bitstrings. Some entropy sources are *conditioned*, meaning that their outputs are expected to provide approximately full entropy, and to be statistically uniform, independent, and unbiased, and thus, in principle, directly usable for cryptographic keys, IVs, etc.

There are two broad types of entropy sources: conditioned and normal (or "raw") entropy sources. Conditioned entropy sources provide bitstrings with full entropy--each bit is unbiased and independent, and a k bit output has k bits of entropy. Normal (or "raw")

4It seems to me that a lot of this section, maybe most of it, could easily end up in Part Two. On the other hand, there are some good reasons to keep it here—it discusses how to take an entropy source that doesn't quite fit the needs of the RBG, and process its outputs so that it does fit. Comments?

5This is nebulous here, but I'm hoping to firm it up....--JMK

entropy sources provide bitstrings with an assessed amount of entropy, but do not try to massage the outputs into any particular distribution; nothing outside the normal entropy source can assume anything about the output distribution, except what is implied by the output size and the assessed min-entropy. Some operations (conditioning, buffering, and the entropy accumulation) may be performed either inside or outside the entropy source. Techniques and guidance for performing these operations inside the entropy source appear in Part Two ~~of this standard~~; constructions for performing these operations outside the entropy source are discussed in this document.

An entropy source must be tested for correct behavior ~~from time to time~~. Entropy sources are nondeterministic, but their behavior is expected to follow some probability model. This means that a deviation from "correct behavior" can be quite subtle and difficult to detect. Some combination of statistical tests based on the probability model used for the entropy source, and testing for known or suspected failure modes~~,~~ can be used to determine, with reasonable assurance, that the source is still behaving properly.

Conceptually, tests of the raw entropy source need to be conducted by the entropy source, although the tests may be implemented externally. The entropy source has detailed information about the probability model of the raw entropy source and has internal values that are not output during normal operation of the entropy source, From the perspective of the RBG design, the goal of these tests is to detect a deviation of the entropy source from its expected behavior that might lead to a critical failure of the source, and thus, to a practical security vulnerability in the application relying upon the RBG.

2.2 Making it fit: Accumulating and Buffering Entropy and External Conditioning

The DRBGs defined in Part 3 have an enormous range of acceptable entropy input sizes; typically, these range up to around four billion bytes--far more than is likely to be useful in practice. However, there are often good implementation reasons to restrict the size of entropy input to some more manageable size; a real-world hardware implementation may not be able to support processing an enormously long string. *External accumulation* of entropy is required when the entropy source is producing long output strings with sparse entropy, which must be condensed into shorter strings to be used; in this case, the entropy source is not performing conditioning on the raw entropy bits to obtain full entropy. External b*uffering* of entropy is required when the entropy source is producing entropy too slowly to meet multiple entropy requests within a short time period in order to store entropy in anticipation of receiving such requests.

2.2.1 Accumulating Entropy

2.2.1.1 Accumulating Entropy Using the Derivation Functions

In practice, a DRBG ~~algorithm~~ uses a derivation function to reshape the output from the entropy source into the size of internal parameters it needs. This is the standard way to map an entropy input to an output of the right size, with uniform and independent bits.

When practical, the derivation functions **should** be used to accumulate entropy into the right size for the DRBGs. It is acceptable to use the derivation function externally from the DRBG, and then to feed the result into the DRBG for instantiation and reseeding. (That is, it is acceptable to call the derivation function to process a long stream of entropy source outputs, generate a result, and then use that result in the DRBG's instantiate or reseed function, even when the DRBG then uses the derivation function a second time to process that input.)

2.2.1.1.1 Using Hash_df to Accumulate Entropy

A hash function is a natural tool for accumulating entropy, and hash_df (as specified in Part 3) provides a reasonable way to use a hash function for this purpose. The requested output size **shall** be a size that the RBG can process, e.g., by buffering or direct use in reseeding or instantiating a DRBG ~~algorithm~~, and **shall** provide at least the number of bits needed for instantiation or reseeding. The output length **should** be a multiple of the hash output size for efficiency. ~~See the full discussion of Hash_df in Part 3 of this standard.~~

So long as the output length is known before processing begins, this can be computed ~~on the fly,~~ without buffering the whole input string. However, when buffering is used, the input string to the hash_df can be buffered using any of the techniques described in Section 2.2.2, or can be used directly.

2.2.1.1.2 Using Block_Cipher_df to Accumulate Entropy

With Block_Cipher_df (as specified in Part 3), it is necessary to specify the input and output lengths before the first block of the input string is processed. For most entropy sources, this is acceptable, as the designer will know how many bits of entropy source output must be processed to provide the required amount of entropy. However, some entropy sources are extremely variable in how much entropy can be produced per bit of output; for those entropy sources, the Block_Cipher_df may be a bad accumulation choice. Note that it is acceptable to pad the input strings with zeroes; the Block_Cipher_df could ~~thus~~ be started with the maximum length of input that might be required, but when sufficient entropy determined to be available in the input, the remainder of the input could be filled in with zeros.

The outputs from the Block_Cipher_df can be buffered using any of the schemes described in Section 2.2.2.

2.2.1.2 Accumulating Entropy in a CRC

A CRC (cyclic redundancy check) is a widely used kind of non-cryptographic checksum that uses a feedback polynomial. A CRC register of U bits can be implemented very efficiently in hardware, and can be used to accumulate entropy from almost any source. The requirement is that there **shall** be no influence of the specific feedback polynomial

on the entropy source's behavior[6]. A CRC **shall** be used as follows:

a. The feedback polynomial for the CRC **shall** be chosen to be irreducible.

b. The CRC **shall** always start at a nonzero value.

c. When an accumulated value is output, the CRC register **shall** retain its value, and the next CRC **shall** be computed starting from that value.

Entropy accumulated in a CRC **should** be buffered using the hash-buffer construction discussed in Section 2.2.2.3, but may use any of the three buffering techniques described in Section 2.2.2.

2.2.1.3 A Software Entropy Accumulation Mechanism

[Note: CRCs provide a good generic way to accumulate entropy in hardware, but are not especially fast in software. ANS X9.82 needs to specify a well-analyzed way to do this, generically, in software. The /dev/random approach looks reasonable enough, but it's hard to prove anything about it. Perhaps a universal hash algorithm would be appropriate here. TBD...]

2.2.2 Buffering Constructions and External Conditioning

The following constructions specify the three allowable ways to buffer entropy externally from the entropy source, for later use by the RBG. (Note that internal buffering (i.e., buffering within the entropy source) is handled in Part 2, and can be much more flexible, as the buffering technique can accommodate the entropy source's probability distribution.) The buffering constructions in this section can be used along with the external entropy accumulation mechanisms described in Section 2.2.1, or can be used directly using the entropy source outputs, subject to the requirements in the descriptions on the different buffering schemes.

This Standard specifies acceptable constructions in order to avoid subtle attacks enabled by designing the buffering scheme in the wrong way. Any practical buffering scheme involves some finite amount of memory, and thus imposes a limit on how much entropy may be collected, but poorly designed buffering techniques could result in failure to use all available entropy produced by the entropy source.

Any of these external accumulating and buffering constructions could be combined with an existing entropy source to provide a new entropy source with somewhat different properties, in terms of the length of its outputs and the rate at which the entropy source can service entropy requests. The XOR Buffer and Hash Buffer constructions in Sections 2.2.2.2 and 2.2.2.3 can also be used to provide a conditioned entropy source. In all cases, all other constructions in Part Four that require an entropy source can use these new entropy source constructions as needed.

6This is true because if I don't tell you my polynomial, you have a very low probabilty of producing a set of outputs that collides more often than expected by chance, at least for reasonable sized inputs.

2.2.2.1 Construction: Simple Queue Buffer

The simplest possible buffering scheme is simply to queue up several of the most recent outputs. This has two major advantages:

a. The scheme is very simple and cheap to implement, and

b. It is ~~obviously~~ no weaker than just using the entropy source outputs as they're produced.

The disadvantage of this buffering scheme is that entropy is never accumulated across outputs; if the entropy source is overestimating its entropy, the simple queue buffering scheme will provide no additional defense, even if the RBG is using entropy from the buffer much more slowly than it is being added to the queue. This affects the validation of the entropy source for use in the RBG if, in normal operation, the RBG is expected to produce many times more entropy than is drawn from the buffer.

Comment [ebb28]: Page: 11
This needs to be explained more, perhaps using an example?

This buffering scheme is applicable to any entropy source, conditioned or not.

The queue contains multiple entries; each entry contains a bit string and an entropy estimate for that bit string. There are a maximum of N entries in the queue. Whenever a new bit string is produced by the entropy source, the bitstring is entered into one end of the queue. If the queue is full, then the oldest entry in the queue is discarded to make room for the newest entry.

When entropy outputs are requested with an indicated minimum entropy, sufficient entries are provided from the end of the queue containing the oldest entries to meet or exceed the entropy requirement. These entries are removed from the queue. When an entropy output is requested, and the queue entries contain insufficient entropy, either an error condition **shall** be raised, or the response to the request **shall** be delayed until there is sufficient entropy in the queue entries to meet the request.

2.2.2.2 Construction: XOR Buffer

A simple improvement to the Queue scheme involves recycling Queue entries that would otherwise be discarded when the buffer is full (see Section 2.2.2.1). This is accomplished by exclusive OR-ing (i.e., XORing) the queue entries to be discarded with the new entries.

When a new entropy source output is entered into the buffer, and the queue is not yet full, the XOR buffer works exactly as does the Queue buffer. When the XOR buffer is full, a new entropy source output is entered into the queue as follows:

a. The oldest queue entry is extracted from the queue to make room for the new entry.

b. A new bit string is created by zero-padding the new or old entropy output as needed until the bit string sizes match, and then XORing them together.

c. A new entropy estimate is created from the old entry's estimate and the new entropy output's estimate. ~~This is done~~ in one of two ways:

Comment [ebb29]: Page: 12
Unless we specifically disallow it, the buffer may contain entries from multiple sources.

(1) If the entropy source outputs are conditioned internally or externally, the new entropy estimate is the sum of the old entry's estimate and the new entry's estimate, or the number of bits in the shorter of the two original bitstrings, whichever is greater. ???]]

(2) Otherwise, the new entropy estimate is the larger of the old and new entry's estimate.

As a full buffer has more entropy added to it, the values are simply XORed together repeatedly, hopefully accumulating more and more entropy until the entire buffer is unpredictable.

When entropy is requested from the buffer, entries are taken from the queue until the number of bits of entropy requested is satisfied. If there is not sufficient entropy in the buffer to satisfy the request, then the buffer must either raise an error condition or delay the response to the request until enough entropy is available. The XOR of all the entries provided for the request is saved and is XORed with the next entropy output entered into the buffer.

Comment [ebb30]: Page: 12
I think that the second way is true in either case. Conditioned entropy bits have full entropy, therefore, the amount of entropy can only be the length of the longest string when the strings are XORed.

Comment [ebb31]: Page: 12
I don't think this is true, see the above comment.

2.2.2.3 Construction: Hash Buffer

A hash buffer will efficiently accumulate entropy for any entropy source in a "pool" of entropy. The buffer consists of R bits, a 32-bit counter C, and a buffer entropy estimate E.

The following **shall** be performed to enter an *inputString* with estimated entropy *ee* into the hash buffer, where **Hash** is the hash function, and *outlen* is the output size of the hash function:

1. $tmp = ee$.
2. While $tmp > 0$:
   2.1 X = **Hash** (C || *buffer* || *inputString*).
   2.2 $C = C + 1$.
   2.3 Shift *buffer* right by *outlen* bits, discarding the rightmost *outlen* bits.
   2.4 Prepend X to the *buffer*.
   2.5 $tmp = tmp - outlen$.
3. $E = \max(E + ee, R)$

The following **shall** be performed to extract a k-bits of entropy from the buffer:

1. If $k > E$: raise an error condition or delay further processing until sufficient entropy is available.
2. tmp = "".
3. While **len** (tmp) < k :

Comment [ebb32]: Page: 13
Shouldn't this be the min?

Comment [ebb33]: Page: 1
According to the next section, we can't say that there is full entropy in the k bits unless E >= 2k. How do we handle this here?

3.1 $tmp = tmp \,||\, \textbf{Hash}\,(C \,||\, buffer)$.

3.2 C = C + 1.

4. $E = E - k$.

5. Return the least significant k bits of tmp.

2.2.2.3.1 Using the Hash Buffer as a Conditioned Entropy Source

The hash buffer may be used as an external *conditioning* routine under the following conditions:

a. To generate k bits of output with full entropy, the buffer **shall** contain at least $2k$ bits of entropy (i.e., $E \geq 2k$).

b. After k bits are pulled from the hash buffer, its entropy estimate **shall** be decreased by $2k$ bits (i.e., $E = E - 2k$).

c. If the hash buffer is used to provide conditioned entropy outputs, it **shall not** also be used to provide normal entropy outputs.

2.3 Combining Sources and Entropy Estimates

In some RBG designs, especially designs based entirely in software, many entropy sources may be combined ~~together~~ to get enough entropy to instantiate a DRBG or to produce full entropy output.

Software entropy sources tend to be wildly variable in the rate of entropy produced, as their unpredictability depends on the natural variability in some process is occasionally used, or which differs in its properties, depending on parameters outside the entropy collection mechanism's control. For example, while hard drive latency is apparently an excellent entropy source, entropy collection code written into the operating system kernel is not likely to *cause* hard drive accesses that intentionally miss all the different levels of cache. When other applications on the system are not accessing the disk, there will be little entropy available from this source. Therefore, both the accumulating and buffering entropy are extremely important in software-based entropy source sysems.

Comment [ebb34]: Page: 1
Is this the right term? Should it be something more like entropy obtained from software interactions or some such?

2.3.1 Accumulating Entropy for Output

Any of the accumulation methods described in Section 2.2.1 are suitable for use with multiple entropy sources. The accumulation of entropy inputs is valuable for very sparse entropy sources (those that produce large numbers of output bits per bit of entropy), such as are often extracted from operating system statistics and system loading sources. When performance requirements permit, the hash_df accumulation method should be used (see Section 2.2.1.1.1).

2.3.2 Maintaining an Entropy Pool

Any of the buffering techniques in Section 2.2.2 may be used with multiple entropy sources. In software, the hash buffer construction can be used to provide an entropy pool, capable of supporting DRBGs, enhanced NRBGs, or composite RBGs

2.3.3 Estimating Entropy from Multiple Sources

An approved RBG requires an entropy source whose assessments are known to be accurate. Entropy estimates used in RBGs **shall** be based only upon assessed entropy from approved entropy sources. When outputs from multiple approved entropy sources are combined, the maximum entropy assessed for the combined outputs **shall** be the sum of the estimates from the approved entropy sources, but may be any value less than or equal to that sum.

Comment [ebb35]: Page: 1
Need to reword if we are allowing anything other than pre-approved entropy sources.

Comment [ebb36]: Page: 1
Doesn't this dewpend on how they are combined?

Unapproved entropy sources may be combined with the approved ones using any of the accumulation mechanisms, as well as the XOR Buffer and Hash Buffer, so long as entropy values from unapproved sources are assessed as having zero entropy.

2.4 Validation Considerations

The hardest problem in constructing a working RBG is in managing entropy so that security is provided even in the face of some kinds of failure of the entropy source. Entropy sources are often quite fragile, and testing them in the field is complicated by the fact that they are not deterministic, and that many failure modes are subtle enough to require computationally expensive statistical tests to detect.

Each RBG construction places somewhat different requirements on the entropy source for validation ~~purposes~~. The entropy source failures that would lead to a critical failure are discussed for each RBG construction (see Sections 3 and 4).

The risk of entropy source failure **shall** be handled in one or more of the following three ways:

a. The entropy source design and implementation minimizes or eliminates some possible kinds of failure. For example, one reference entropy source from Part 2 uses three ring oscillators with different average periods. This design substantially reduces the risk of failure from the phase locking of the oscillators with some stable on-chip signal, because one signal will generally not be able to phase lock with more than one of these ring oscillators. Alternatively, an implementation might carefully shield the entropy source from any external signals to avoid the risk of an oscillator phase locking with some external signal. Part Two of this Standard includes an extensive discussion of this kind of design and implementation decision.

b. The entropy source has substantial continuous or periodic testing to detect likely failure modes. For example, an entropy source based on counting Geiger counter clicks might run a Chi-square test on the counts it for a run of 1000 samples

during startup testing, and verify that this is consistent with some Poisson parameter for the count distribution within the design specifications of the device. Part Two ~~of the standard~~ discusses the etails of testing at different points for entropy sources.

c. The surrounding mechanisms that use the entropy source ~~can~~ provide some level of overdesign, so that minor or short-duration failures in the entropy source will have minimal impact. For example, a DRBG might require twice as much estimated min-entropy in the input it uses from the entropy source to instantiate for the first time, and thereafter rely on the persistent state to retain enough entropy to protect against incorrect entropy estimates from the source. This is discussed for each of the RBG constructions, below.

A failure in the entropy source that is not detected or prevented from doing any harm leads to a *critical failure*, and thus introduces a practical security vulnerability. A major design goal for any RBG is to keep the probability of a critical failure to an acceptably small value, and a major goal of the validation of an RBG is to verify that this probability is acceptably small.

There are three kinds of entropy source failures that can be critical failures, depending on the RBG that uses the entropy source.

a. A failure during the first time that the entropy source is used undermines all security for an Externally Seeded DRBG, and can lead to a critical failure in Internally Seeded DRBGs with seed files.

b. A failure immediately after startup can lead to a critical failure in Internally Seeded DRBGs and Enhanced NRBGs, when those RBGs do not have seed files.

c. A failure at any other time during operation can lead to a critical failure in Basic NRBGs.

Comment [ebb37]: Page: 1
installation?

Comment [ebb38]: Page: 1
For any DRBG, unless sufficient additional entropy is added before any bits are output.

Comment [ebb39]: Page: 1
Haven't discussed seed files yet. Actually, as long as the instantiated entropy is still availble (whether or not there are seed files), it's still OK (but prediction resistance cannot be provided).

3 Building a Computationally-Secure RBG (DRBG)

A Deterministic Random Bit Generator mechanism (hereafter called a DRBG) provides computational security.

The basic problem of such a mechanism is instantiating the underlying DRBG ~~algorithm~~ securely--reaching a secure starting point, from which outputs can be generated that are computationally indistinguishable from ideal random bitstrings. The DRBG may also support requests to reseed, perhaps in response to a request for prediction resistance, if there is a live entropy source available.

Every RBG containing a DRBG requires some source of an unpredictable internal state, and some algorithm for generating secure cryptographic pseudorandom bits from any unpredictable internal state. There are two constructions for such RBGs:

a. DRBG with a Live Entropy Source

Comment [ebb40]: Page: 16
Would it be useful to define computational security, or tell why a DRBG is in this class?

b. DRBG without a Live Entropy Source

These constructions are discussed below. Note that the RBG can be built with no live entropy source by using persistent storage and at least a one-time access to an external entropy source or other RBG (case a), or with an entropy source but no persistent storage (case b), or with both. There is no approved construction for an RBG without either persistent storage or a live entropy source.

Comment [ebb41]: Page: 17
We'll need to define this; e.g., storage that retains its values for a long time, especially over losses of power?

3.1 Preliminaries

A DRBG promises at least s bits of security, where s is the security level. This means that distinguishing the output sequence of the DRBG from random bits should be no easier than distinguishing the output sequence from an ideal block cipher running in counter mode with an s-bit key from random bits. Any consuming application with an s-bit or lower security level may use a DRBG with an s-bit security level.

Comment [ebb42]: Page: 17
Does it have to be counter mode, or is this an example mode?

The most important problem in using any DRBG ~~algorithm~~ is getting to a *secure state*--a value of the working state from which outputs can be generated that will satisfy the k-bit security level claimed by the design. In practical terms, this means that the secret part of the working state must be *unguessable* by the attacker. A secondary problem is ensuring that the DRBG recovers from compromise whenever possible. That is, if the working state should somehow be leaked or learned through cryptanalysis, the DRBG should eventually reach a new secure state if it is provided access to more entropy over time.

All DRBGs in Part 3 specify functions for instantiating and reseeding the DRBG as well as for generating pseudorandom bits.

a. The *instantiate* function obtains *entropy_input* and combines it with an optional *personalization_string* to instantiate the DRBG,

b. The *reseed* function obtains entropy input and combines it with the current entropy in the working state and optional *additional_input* to produce a new working state, and

c. The *generate* function uses the current working state and optional *additional_input* to produce both pseudorandom bits and a new working state.

That is, the DRBGs use whatever entropy is in the *entropy_input*, *personalization_string*, and *additional_input* parameters to reach a secure state.

A DRBG requests entropy from its entropy source when instantiating the DRBG and when reseeding it, perhaps in response to a request for prediction resistance. The goal for each of these ~~three~~ uses of entropy is to obtain a secure state for the DRBG, even if the DRBG was completely compromised before. The DRBG algorithms are designed to ensure that even entropy inputs under the control of an attacker cannot force an already secure DRBG into an insecure state.

3.1.1 Instantiation

The Instantiate function takes two inputs: The entropy input, and the personalization string. The goal of the entropy input is to provide a value that cannot be guessed by any attacker, so that the DRBG will end up in an unguessable and secure state after instantiation. The goal of the personalization string is to provide a unique input value for each DRBG implementation, and ideally, for each DRBG instantiation.

For the validation of a DRBG claiming k bits of security, the most important questions are: Can the attacker guess the whole input to the instantiate function with much less than 2^{k-16} work? And will an unknown input to the instantiate function ever repeat? If the answer to either question is "yes," then the DRBG **shall not** pass validation.

Comment [ebb43]: Page: 18
Explain where this value comes from.

There are two ways to ensure the answer to both questions is "no":

a. If the entropy input contains the required k+64 bits or more of min-entropy, then guessing the inputs to the instantiate function is intractible and there is a negligible chance of repetition of the input. A DRBG design can improve the chances of this happening by *oversampling*—requesting some multiple of the minimum entropy required for instantiating the DRBG. In the remainder of this document, the term *oversampling factor* is used to describe this multiple, and the symbol w is used.

b. If the entropy input and personalization string together contain at least k + 64 bits of min-entropy, and never repeat, then the DRBG is instantiated into a new, secure state each time the instantiate function is called. A DRBG can improve the chances of this happening by guaranteeing that the personalization string contains a secret of at least k + 64 bits of min-entropy, and also a value that will never repeat for two instantiations, such as a timestamp or a monotonic counter value.

Comment [ebb44]: Will need to reflect this in Part 3.

Comment [ebb45]: Page: 18
Need to discuss why the entropy is required in the personalization string rather than the entropy input (I know this is not really true, but the case needs to be made for recommending teat the personalization string contain this much entropy).

Either or both of these techniques may be used by a DRBG to minimize the chances of a critical failure.

The different RBG constructions have very different ways of minimizing these chances, and different resources with which to do so.

3.1.1.1 Entropy Input and the Oversampling Factor

Comment [ebb46]: This needs to be in Part 3.

The RBG design can specify an *oversampling factor*, w. An RBG design with an oversampling factor of w instantiates its DRBG with at least w times the number of bits of min-entropy required by the algorithm. This can be done in two ways:

a. The DRBG algorithm can be reseeded w times, in the normal way, before any outputs are generated. This is the only way to apply an oversampling factor when using a DRBG that is instantiated with full entropy inputs when a derivation function is not used.

b. When a derivation function is available, the DRBG algorithm can be reseeded with

a string that has w times as many bits as are minimally required for the instantiation. ~~This amounts to a change in the Get_entropy call discussed in Part Three during instantiation.~~

Note that in the DRBG construction without a live entropy source, the notion of an oversampling factor is meaningful only for its seed file initialization, not for instantiating its DRBG.

Comment [ebb47]: Page: 19
Seedfiles are not discussed yet.

3.1.1.2 Personalization Strings

A personalization string is optional, but **should** be used. The following requirements and recommendations apply, but see Part Three for a much more in-depth discussion of personalization strings.

a. A personalization string **shall** be expected to be unique to a single implementation or device. For example, it might contain a device serial number.

Comment [ebb48]: Page: 19
user?

b. A personalization string **should** be expected to be unique per instantiate request. For example, it might contain a timestamp accurate to hundredths of a second.

c. A personalization string may contain secret information. Secret information that requires protection from disclosure at a higher security level than the DRBG supports **shall not** be included in the personalization string.

Note that personalization strings are entirely optional; some DRBG designs will not include them, though this has an impact on validation.

Comment [ebb49]: Page: 19
Don't understand this.

3.1.2 Generation of Bits

All the DRBG constructions handle the generation of bits by calling their underlying DRBG algorithms. Any additional requests, such as prediction resistance, are passed along to the underlying DRBG algorithm, if they are supported.

3.1.3 Reseeding

Reseeding is handled differently in the different DRBG constructions. DRBGs without a live entropy source are not likely to ever reseed, and the reseed function may be omitted from an implementation. For DRBGs with live entropy sources, reseeding strategy is a major issue, as discussed below.

3.1.4 Implementation Validation Concerns

An RBG containing a DRBG **shall** NOT pass validation if ~~and only if~~ the probability of a failure to instantiate the DRBG to a secure point is acceptably low.

3.2 Construction: RBG with an Internally Seeded DRBG

An RBG with an internally-seeded DRBG consists of DRBG functions and an entropy

source. This is the simplest DRBG construction, and if implemented properly, can be quite secure.

Comment [ebb50]: Page: 20
An externally seeded DRBG could be considered simpler, at least by the implementer.

3.2.1 Instantiating the DRBG ~~Algorithm~~

Before the first bit of output is generated from the DRBG, the DRBG ~~algorithm~~ must be instantiated. The DRBG instantiate algorithm requires two parameters: the entropy input and the personalization string.

The DRBG design specifies an oversampling rate, w. This rate is used to determine the minimum amount of entropy required before the DRBG can be used to generate bits.

[Discuss the two ways to implement: having a reliable entropy source available whenever instantiate or reseed is required, or using a seed file for the case where the entropy source may not always be available, and the sedfile is used as a failsafe mechanism.]

3.2.1.1 Saving Entropy Across Startups: The Seed File

Comment [ebb51]: The seed file discussions need to be included in Part 3, and teh procedures need to accommodate them.

Some DRBGs ~~(especially all-software ones)~~ may have entropy sources that are not reliably available in time to instantiate the DRBG ~~algorithm~~ before it is needed. Other DRBGs may have an entropy source that could fail in the field, and for which there is not sufficient time at startup to do an extensive battery of statistical tests to detect the failure. In these cases, the DRBG can save some entropy across startups, using a *seed file*. (Note that the use of the word *file* here is not meant to imply any particular way of storing the entropy.) A seed file **shall** be stored in persistent nonvolatile memory.

Comment [ebb52]: Page: 20
Prior to the first instantiation? prior to the first use of the DRBG to generate bits? prior to each instantiation after the first? Between reseeds?

Comment [ebb53]: Page: 20
instantiations? reseeds?

Comment [ebb54]: Page: 20
Discuss this concept and its implications.

3.2.1.3.1 Instantiating with the Seed File

The seed file is processed as follows during each instantiation:

a. The entropy source is used to generate an entropy_input.

b. The personalization_string is constructed, using whatever personalization information is available, plus the current contents of the seed file.

Comment [ebb55]: Page: 20
Need to indicate that the first acquisition of the entropy input and personalization string MUST contain sufficient entropy fso that the first seedfile will thereafter contain sufficient entropy.

c. The DRBG algorithm is instantiated.

d. ***Before any other outputs are generated***, the DRBG algorithm generates *security_level+64* bits of output, which are written to the seed file, overwriting any previous values.

3.2.2 Generating Bits

A DRBG implementation may support output generation with prediction resistance. Generate requests to the DRBG are simply translated to requests to the underlying generate algorithm.

3.2.3 Reseeding the DRBG ~~Algorithm~~

Reseeding a DRBG ~~algorithm~~ means adding entropy input (and *additional_input*) into the

DRBG internal state, in such a way that if the DRBG was not previously in a secure state, it ends up in one, while if it started in a secure state, even full attacker control over all the inputs cannot force it into a weak state.

3.2.3.1 When Should a Reseed Occur?

The DRBG ~~algorithm~~ may be reseeded for four reasons:

a. A consuming application specifically requests reseeding based on a user's decision.

b. Some consuming application asks for bits with prediction resistance.

c. The DRBG ~~algorithm~~ reaches a limit on its outputs, which requires reseeding before more bits may be generated.

d. The DRBG carries out a reseed based on available entropy and its *reseeding strategy* to minimize the exposure due to any compromise of the DRBG internal state (see Section 3.2.3.3).

The DRBG requests at least k bits of min-entropy from its entropy source (where k is the *security_level*), and provides this as entropy input to reseed the DRBG ~~algorithm~~. Additional information that is likely to be different each time the DRBG reseeds, such as timestamps, other system status information, or seed file contents (see below) may be included in the *additional_input* to the reseed function.

3.2.3.2 Seed Files and Reseeding

Comment [ebb56]: Needs to be accommodated in Part 3.

If a seed file is present, it **shall** be used in reseeding in the following way:

a. The seed file contents are included as part of the *additional_input* to the reseed function

b. Immediately after reseeding, the DRBG generates *security_level*+64 bits of output. This output string is used to overwrite the previous seed file contents.

3.2.3.3 Reseed Strategy

3.2.3.3.1 Using Entropy Wisely in a DRBG

~~Most~~ DRBGs with an entropy source available may have far more entropy available than is required to operate the DRBG, after the startup. By forcing reseeds to occur from time to time, the DRBG can attempt to protect the internal state from as-yet-unknown cryptanalytic attacks, as well as from any other form of compromise.

Comment [ebb57]: Page: 21 initial instantiation?

The RBG collects and buffers entropy from the entropy source over time. At some point, the RBG triggers a reseed of the DRBG ~~algorithm~~, based on the amount of entropy buffered, the number of outputs since the last reseed, the amount of time since the last reseed, and potentially any number of other factors.

There are two goals for reseeding:

a. By waiting until a large amount of entropy is buffered, the RBG can give the DRBG ~~algorithm~~ the best possible chance of eventually getting to a secure state, even in the face of a serious overestimate of entropy from the entropy source.

b. By reseeding often, the RBG can give the DRBG ~~algorithm~~ the best possible chance of recovering from compromises of its internal state, and of resisting unknown cryptanalytic attacks.

3.2.3.3.2 Reseeding Before the First Output is Generated

Any RBG with a DRBG with the capability of buffering the live entropy source **should** implement the following procedure:

a. The RBG collects and buffers entropy from the entropy source until the first DRBG output is requested.

b. The full contents of the buffer are used to reseed the DRBG ~~algorithm~~.

c. The DRBG ~~algorithm~~ responds to the request.

This gives the DRBG ~~algorithm~~ its best possible chance of getting a new secure state before the first output is generated.

If no facility is available for buffering entropy from the entropy source, the DRBG ~~algorithm~~ **should** implement the following procedure: Reseed the DRBG ~~algorithm~~ as many times as time and other resources permit, before the first output.

This is the most valuable and important piece of the reseedstrategy, and **should** be implemented even if all other reseed advice is ignored.

Comment [ebb58]: Page: 22
What other reseed advice is there to be ignored?

3.2.3.3.3 Reseeding After the First Output

After the first output, the proper reseed strategy depends on the system designer's assessment of threats. Frequent reseeds provide substantial practical protection from crytpanalysis of the DRBG algorithm, in exactly the same way as frequently changing the key of any cryptographic algorithm provides protection against cryptanalysis. Reseeds that take place only after accumulating sufficient entropy in accordance with the security level provide a chance to recover from compromise even in the face of serious failure of the entropy source~~, so long as *some* entropy is coming from the source~~.

Comment [ebb59]: Page: 22
This seems to conflict with the concept of providing prediciton resistance. Perhaps this needs to be removed or more information needs to be provided.

A simple reseed strategy that addresses these goals is as follows:

The DRBG maintains two accumulations of entropy, called the *fast pool* and the *slow pool*, with alternating entropy outputs going into each pool. Reseeds are then triggered as follows:

a. Whenever the fast pool's assessed entropy is k bits, the DRBG uses its contents to reseed the DRBG ~~algorithm~~.

b. If a reseed is triggered by the DRBG ~~algorithm~~ or the consuming application, the fast pool's contents are used, along with whatever additional entropy source outputs are required to reach an assessed value of at least k bits of min-entropy for the reseed.

c. The slow pool's reseed threshhold starts at $t = 2*k$. Whenever the slow pool reaches t bits of assessed entropy, the DRBG does the following steps:

(1) Reseed the DRBG ~~algorithm~~ with the contents of the slow pool.

(2) Sets a new reseed threshold to $t = 2*t$, unless t is already at its maximum value. (The practical advantages of this design fall off as t gets very large, e.g., $t > 256$.)

More elaborate reseed strategies are possible~~; this one is based on a combination of the ideas from Yarrow-160 and Fortuna~~.

Comment [ebb60]: Page: 23
Need to discuss what this means.

Comment [ebb61]: Page: 23
256 is not a very big number. Do you mean 2^{256}?

Comment [ebb62]: Page: 23
Since the above seems to be an example, what are the requirements that we can state, if any?

3.2.4 Implementation Validation Concerns

Validation must answer the question: Is the probability that the DRBG ~~algorithm~~ will reach a secure, unguessable state before the first input acceptably high, even in the face of possible entropy source failures?

3.3 Construction: Externally Seeded DRBG

A DRBG can implemented without access to a live entropy source. This DRBG cannot support automatic reseeding, including the ability to generate outputs with prediction resistance. An externally seeded DRBG **shall** have a fixed cryptoperiod, which may be stated in terms of maximum number of outputs allowed, or of maximum time until the DRBG must be discarded, or re-instantiated or reseeded using an external source of entropy input. Note that the differences between reseeding and re-instantiating are as follows:

a. Reseeding requires an additional function that is slightly different than the instantiate function, and

b. The reseed function uses both new entropy bits and entropy contained within the internal state of the instantiation to be reseeded, whereas the instantiate functin uses only the new entropy bits.

The Externally Seeded DRBG provides a less convenient, and less secure alternative to an internally seeded DRBG, but may be more affordable in some environments.

The only entropy available to this DRBG is kept in the seed file (if used), in some secure, persistent storage. If the storage is manipulated or read by an attacker, the DRBG must be assumed to irretrieveably lose all security.

3.3.1 First Instantiation: The External Seeding

The Externally Seeded DRBG is instantiated using entropy input obtained from outside the RBG. There are two permitted methods of doing this:

a. *State Importation: An* external, trusted source may provide the DRBG with a complete DRBG working state for some DRBG ~~algorithm~~. In this case, the DRBG implementation may omit support for the Instantiate function.

b. *Entropy Importation:* An external, trusted source may provide the entropy input for the DRBG, which then instantiates using its own routines.

3.3.1.1 Requirements for State Importation

The following requirements apply to First Instantiation:

a. The instantiate function **shall** be trusted and subject to validation.

b. The instantiate function **shall** have a validated Internally Seeded DRBG, including the Instantiate function for at least the security level claimed by the target Externally Seeded DRBG.

c. The instantiate function **shall** use its entropy source ~~and Instantiate function~~ to produce a securely instantiated DRBG's working state.

d. The working state produced above **shall** be imported into the Externally Seeded DRBG over a secure and authenticated link of no less security level than the Externally Seeded DRBG claims.

e. The DRBG boundary containing the instantiate function **shall not** retain the working state generated, or the entropy input from which it was instantiated.

f. Immediately upon importing the working state, the Externally Seeded DRBG **shall** perform the following steps[7]:

(1) Let S_0 = the transferred working state.

(2) Generate a single output of one bit and another working state S_1 using the Generate function, from the ~~current~~ working state S_0 and *additional_input* = "X".

(3) Store the new working state S_1 in persistent storage.

(4) Generate a single output of one bit and another working state S_2 using the Generate function, from the ~~current~~ working state S_1, with no *additional_input*. Discard the output.

Comment [ebb63]: Page: 24
This is the distributed DRBG boundary case.

Comment [ebb64]: Page: 24
This needs to be modified to address the transfer of internal state values between DRBG boundaries.

Comment [ebb65]: Page: 24
This needs to be recast in light of the distributed DRBG.

Comment [ebb66]: Page: 24
Need to recast this is light of a distributed DRBG and the security levels in FIPS 140-2. However, this needs to be done without referring to FIPS 140-2, I guess.

Comment [ebb67]: Page: 24
What if the DRBG containing the instantiate function also contains a reseed function (ugly as this seems)? In this case, the current working state needs tp be transferred back to the reseed function. Or should we just disallow this, since it's just too complicated?

Comment [ebb68]: Page: 24
Recast for a distributed DRBG.

Comment [ebb69]: Page: 24
Do we need to indicate that S2 may/should also be placed in persistent storage. Some explanation seems to be missing here.

7This ~~song-and-dance routine~~ guarantees that outputs are not repeated if the system crashes after instantiating.

3.3.1.2 Requirements for Entropy Importation

Requirements associated with the importation of entropy input for instantiation are:

a. The external source **shall** be trusted and subject to validation.

b. The external source **shall** be an approved entropy source or an approved RBG with at least the same security level as that claimed by the Externally Seeded DRBG.

c. The external source **shall not** retain the entropy input provided to the Externally Seeded DRBG.

d. The Externally Seeded DRBG **shall** carry out its Instantiate function using the entropy input provided by the external source, and a personalization string **should** be provided during instantiation.

e. Immediately upon instantiation, the Externally Seeded DRBG **shall** generate a *security_length*+64 bit pseudorandom output, and store this in its seedfile if a seedfile is supported by the design.

f. Entopy input **shall** be entered and handled in the same manner as a cryptographic key of the same security level.

Comment [ebb70]: Page: 25
Frankly, this is hard to mandate. For validation purposes, the vendor could provide guidance; but it's teh user's responsibility to provide teh entropy input in this case. Coin flipping would be hard to validate, for instance, though the vendor could provide guidance on the procedure.

Comment [ebb71]: Page: 25
See above comment. This is a user requirement.

Comment [ebb72]: Page: 25
A user requirement.

Comment [ebb73]: Page: 25
This is user guidance, except the the implmentation should support it.

3.3.2 Instantiation Before the First Output is Generated

Comment [ebb74]: Page: 25
We probably need to cover both "prior to first output (installation?) as well as repeated power ups.

Prior to the first use of the DRBG to generate bits, the Externally Seeded DRBG must be instantiated with sufficient entropy for the intended security level. There are two approaches possible for this, depending on whether the Instantiate function is supported by the DRBG

3.3.2.1 ~~Power Up with~~ Instantiation Support and a Seed File

Comment [ebb75]: Page: 25
Power up may not be the best term. A seed file is optional, depending on the availability of the entropy input, etc.

With a seedfile and support for Instantiation, the following process is used:

a. The Externally Seeded DRBG is instantiated, using the contents of the seed file as the entropy input. The Externally Seeded DRBG **should** use a personalization string, if one is available.

Comment [ebb76]: Page: 25
A seed file doesn't exist the first time.

b. ***Immediately upon instantiation***, the Externally Seeded DRBG **shall** generate an output of *security_level*+64 bits, and write that output to the seed file, overwriting any previous value.

Comment [ebb77]: Page: 25
A seed file dosen't exist the first time, but can be used for repeated power ups.

3.4.2.2 ~~Power Up~~ Without Instantiation Support

Comment [ebb78]: Page: 25
Should the instantiate function in Part 3 include this?

Comment [ebb79]: Somewhere, there needs to be a distinction made between a power up that is maintained, and a concept of multiple power ups.

When the Externally Seeded DRBG does not have Instantiate support, the following process is used at power up:

a. The ~~current contents of~~ working state is read into memory as the working state S_0.

b. Generate a single output of one bit and a new working state S_1 using the Generate function from working state S_0 and *additional_input* = "X".

Comment [ebb80]: Page: 25
This seems to be the distributed DRBG boundary case.

Comment [ebb81]: This needs to be accommodated in Part 3?

c. Store working state S_1 in persistent storage.

d. Generate a single output of one bit and a new working state S_2 using the Generate function, from the current working state, with no *additional_input*. Discard the output.

Comment [ebb82]: Page: 26
As before, do we need to indicate that S2 may/should also be placed in persistent storage. Some explanation seems to be missing here.

3.3.3 Generating Bits

The Externally Seeded DRBG generates outputs by calling the ~~underlying~~ DRBG's Generate function. The reseed limit parameter is set to enforce the cryptoperiod of the Externally Seeded DRBG.

This DRBG construction cannot support prediction resistance, but may support additional input. If so, additional input is handled as follows, to ensure that the Externally Seeded DRBG benefits from any entropy that is provided to it:

Comment [ebb83]: Page: 26
Conditions are missing.

3.3.3.1 Processing Additional Input when a Seed File is Supported

Comment [ebb84]: This needs to be accommodated in Part 3.

When the DRBG supports ~~Instantiation and has~~ a seed file, the following process is done immediately before the Generate function generates pseudorandom bits to be returned to the consuming aplication (but before they are actually returned):

a. Generate *security_level*+64 bits of pseudorandom output from the DRBG.

b. Overwrite the current contents of the seed file with these newly generated output bits (i.e., not the bits to be returned to the consuming application).

Comment [ebb85]: Page: 26
Shouldn't teh generate functin in Part 3 include this?

3.3.3.2 Processing Additional Input when a Seed File is not Supported

Comment [ebb86]: This needs to be accommodated in Part 3.

When the Externally Seeded DRBG does not support ~~Instantiate or have~~ a seedfile, the following process is done immediately before the Generate function generates pseudorandom bits to be returned to the consuming aplication (but before they are actually returned):

a. Let S_i = the working state.

b. Generate a single output of one bit and a new working state S_{i+1} using the Generate function from working state S_i and *additional_input* = "X". Discard the output.

c. Store working state S_{i+1} in persistent storage.

d. Generate a single output of one bit and a new working state S_{i+2} using the Generate function from working state S_{i+1} and no *additional_input*. Same question as above.

Comment [ebb87]: Page: 26
This is something that would need to be done by the generate function, it hasn't been specified yet.

3.3.4 Implementation Validation Concerns

Validation of this construction consists of answering the following questions:

a. Is the DRBG securely instantiated the first time?

b. Can an attacker or a component failure cause the persistent storage or seed file used by the Externally Seeded DRBG to be disclosed to some attacker, to repeat values, or to take on predictable values?

Comment [ebb88]: Page: 27
Need to address the case where the seed file is not used.

4 Information-Theoretically Secure RBGs: NRBG Constructions

An NRBG produces bits that are indistinguishable from random, even given unlimited computing power. In other words, the bits output from an NRBG are close enough to being ideal random bits in terms of distribution, that even given a large number of output bits, there is not sufficient information available in the output sequence to distinguish it from an ideal random sequence.

4.1 Preliminaries

Information theoretic security can be provided only by the entropy source. Since entropy sources are generally a lot less reliable than deterministic components, a fundamental question to ask about this kind of construction is "what happens when the entropy source fails in some undetected way?" There are two broad NRBG designs:

a. *Enhanced NRBGs* provide a fallback to an approved DRBG if the entropy source suffers some kind of disasterous failure. This has the practical effect of making the validation of the entropy source much easier.

b. *Basic NRBGs* make no ~~guarantee of a~~ fallback to an approved DRBG if there is a failure of the entropy source. This means that validation and health testing of the entropy source must be much more demanding.

4.1.1 Conditioning

All NRBGs claim information theoretic security, which is possible because bits are obtained from a live entropy source. A fundamental part of any NRBG is "conditioning" the bits from the entropy source, i.e., mapping unpredictable bits from the entropy source to uniform, independent, random bits with full entropy for output from the NRBG.

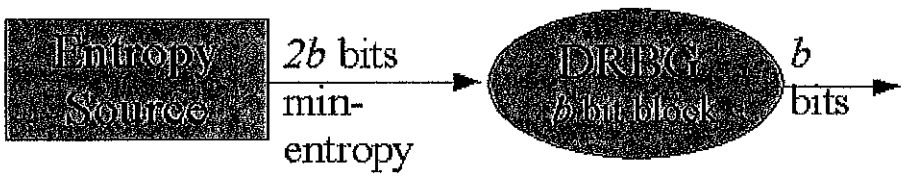
Techniques are provided in Section 2~~, above, provide the ability~~ to externally condition any entropy source. The Basic NRBG with External Conditioning construction discussed in Section ??? uses these techniques, and the Enhanced NRBG Based on XOR xonstruction discussed in Section 4.3 can use them to obtain a conditioned entropy source.

The external conditioning of an entropy source can make no assumptions about the internal probability model of the entropy source, but must assume that the assessed entropy from the source is correct. Any external conditioning technique requires at least $2k$ bits of min-entropy as input, in order to reliably produce k bits of full-entropy output. As discussed in Part Two, techniques for conditioning inside the entropy source can make

use a detailed knowledge of the probability model, and thus can be much more efficient.

The Enhanced NRBG Based on continuous Reseeding (see Section 4) essentially uses an approved DRBG to condition an entropy source. It is acceptable, though somewhat wasteful, to use a conditioned entropy source with this construction.

4.2 Construction: Enhanced NRBG Based on Continuous Reseeding[8]



A natural way to build an enhanced NRBG is to use an approved DRBG to essentially condition the entropy source. The entropy source provides the unpredictability, and the DRBG provides both conditioning in the presence of sufficient input entropy, and computational security in its absence.

Each DRBG has a natural *blocksize*—the number of output bits that can are generated by the underlying cryptographic primitive at a time. For example, a block cipher-based DRBGs that uses AES as the cryptogrpahic primitive has a blocksize of 128 bits, while a hash-based DRBG using SHA256 has a blocksize of 256 bits.

Each DRBG also has a seed length (*seedlen*) appropriate to its cryptographic primitive.

In order to generate full-entropy outputs, the output size b for each DRGB and cryptographic primitive is defined as:

$b = min(blocksize, seedlen/2)$

Comment [ebb89]: Page: 1
What is the rationale for seedlen/2?

4.2.1 Components of the NRBG

The enhanced NRBG consists of two components:

a. A DRBG ~~algorithm~~ and associated internal state from Part 3, to be used with output size b as discussed above.

b. An approved entropy source from Part 2.

8I think I could redesign this NRBG construction to use the DRBG only via an envelope or one of my previous DRBG constructions, but it would make the NRBG itself somewhat less efficient. Let's discuss whether this would be a better way of doing things.

4.2.2 Instantiation

Sufficient entropy for instantiating the DRBG is critical to maintianing the security of the NRBG, even when the entropy source partially fails. As a result, the DRBG **shall** be instantiated using an oversampling multiple $w \geq 2$ ~~of the minimum number of bits of entropy required by the DRBG algorithm,~~ as discussed in Section 3~~, above~~, and **should** satisfy $w \geq 4$. This oversampling multiple can be applied by increasing the amount of entropy requested for instantiation by a factor of w, or by reseeding w times.

A seed file, as discussed in Section 3 ~~above~~, may be used during instantiation.

A personalization string **should** be used, and may include a long term secret value of at least *security_level* bits~~, where k is the claimed security level of the DRBG algorithm.~~

Comment [ebb90]: Page: 1
The security level depends on the crypto primitive and the amount of entropy requeted during instntiation; there is a maximum security level possible, however

4.2.3 Generation of Outputs

When the Enhanced NRBG receives a request for n bits of full-entropy output, the following process takes place[9]:

1. *tmp* = "".
2. While len (*tmp*) < *n*:
   2.1 *seed* = at least $2b$ bits of min-entropy from the entropy source.
   2.2 *tmp* = *tmp* || Generate (b,*seed*)
3. Return the leftmost *n* bits of *tmp*.

Comment [ebb91]: Page: 1
We've discussed that this is not the way to do this because of the reseed interval.

4.2.4 Reseed Management

Comment [ebb92]: Page: 1
Reword based on changes to 4.2.3.

Explicit reseeding the DRBG ~~algorithm~~ MAY be done at any time, but reseeds are not required. Note that the process of generating outputs always inserts more entropy into the DRBG internal state than is taken out, so that the DRBG's internal state is constantly being replaced with a new, unpredictable internal state. One way to effectively reseed the DRBG is to generate more than *seedlen* bits of NRBG output, and discard the output[10].

4.2.5 Making a Composite RBG

This construction ~~described here~~ can be extended to allow users access to both computationally-secure and information-theoretically secure bit strings. This is very valuable when the device is servicing both security-critical operations like key pair generation, and relatively low-importance operations like generating nonces, TCP sequence numbers, IVs, etc.

9I'm trying to figure out if Dan's attack requires us to add a reseed at the end, here. Should we just do the whole thing with reseeds?

10This works for everything but Hash_DRBG; I'm not sure about Hash_DRBG, though.

To provide n bits of computationally secure output ~~bits~~, a request is made to the DRBG's Generate function ~~is called~~ to produce n bits with prediction resistance[11]. ~~With direct access to the algorithm, this is done as~~ follows:

a. Let *seed* = a string with at least k bits of min-entropy from the entropy source.

b. Reseed the DRBG algorithm with entropy_input = *seed*.

c. Call the DRBG algorithm's generate method to satisfy the request for n pseudorandom output bits.

Comment [ebb93]: Page: 1
Hopefully, this can be accomplished by accessing the using the regular generate call.

Note that while the NRBG outputs can support any security level, the DRBG ~~components~~ cannot support any security level higher than that of the DRBG primitive can support.

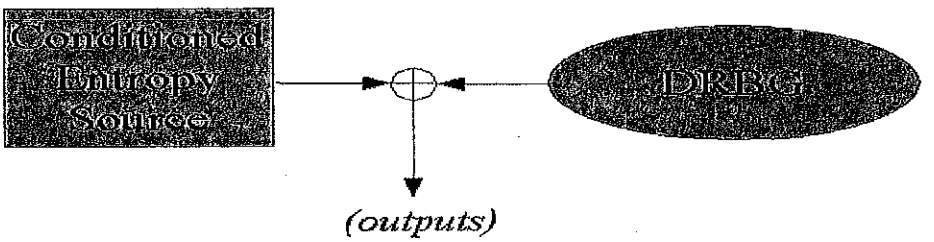
4.2.6 Validation Concerns

Validation must ensure that the DRBG reaches a secure state before the first output is generated, and is thus the same as for the DRBG with Internal Seeding or DRBG with Internal Seeding and Memory constructions, depending upon whether there is a seedfile available.

Comment [ebb94]: Page: 1
Is this really dependent on the use of a seed file?

4.3 Construction: Enhanced NRBG Based on XOR[12]

Given a conditioned entropy source and a DRBG, this NRBG construction works by 1) instantiating the DRBG ~~algorithm~~, 2) satisfying each k-bit request for full-entropy output by generating k bits from the DRBG ~~algorithm~~ and k bits from the conditioned entropy source, and 3) XORing the k-bit outputs together. This very cleanly provides assurance of security that is no worse than the stronger of the two sources of random bits (i.e., the conditioned entropy source and the DRBG), and is a special case of the construction in Section 5.1 for combining RBGs.



11This is required, or a computationally powerful attacker can determine the DRBG state from the outputs seen, and then backtrack to the previous state, because backtracking is only guaranteed to be as strong as the DRBG.

12This Enhanced NRBG can be done using only Elaine's DRBG envelopes with no loss of efficiency or added complexity—one of many much cleaner aspect of the design.

4.3.1 Instantiation

Instantiation proceeds exactly as in Section 4.2.1. That is, the DRBG algorithm is instantiated with at least *w(security_level+64)* bits of min-entropy, where *security_level* is the claimed security level of the DRBG.

4.3.2 Generation

A request for n bits of NRBG output is satisfied as follows:

a. Let $T = n$ bits of output from the conditioned entropy source.

b. Let $U = n$ bits of output from the DRBG ~~algorithm~~.

c. Return T ⊕ U

4.3.3 Reseed Management

In this design, the DRBG does not receive any entropy from the entropy source in normal operation. Designs using this construction **should** reseed the DRBG when there is entropy available that would otherwise go unused, and **shall** respect the reseed limits of the DRBG ~~algorithm~~.

4.3.4 Using the Construction as a Composite RBG

~~In order to provide access to n bits of computationally-secure RBG outputs, the DRBG's~~ Generate function is used to generate n bits of computationally-secure outputs, which are then returned.

4.3.5 ImplementationValidation Concerns

As with the ~~previous~~ Enhanced NRBG construction based on continuous reseeding in Section 4.2, the requirement on validation is simply that the DRBG ~~component of the construction~~ reaches a secure state before the first output is generated.

4.4 Basic NRBGs

In this Standard, a *Basic NRBG* is defined as an RBG that claims information-theoretic security, but does *not* provide security in the event of an entropy source failure. This ~~kind of~~ construction falls into three broad categories, with different design goals behind each category.

a. Extremely lightweight NRBGs can be designed, using very few gates and very little power. Some applications may benefit from having an approved RBG that does not require any cryptographic operations, and whose supporting hardware is measured in hundreds of gates at most.

b. NRBGs can designed to provide high security using only non-cryptographic

processing and well-understood and well-analyzed entropy sources. Both the design and the validation of such an NRBG are quite demanding.

c. Enhanced NRBGs can be designed that default to some non-approved DRBG for their security. Existing hardware random bit generators made before this Standard was published are a natural source of this category of NRBG.

4.4.1 Validating a Basic NRBG

In the previously discussed NRBG constructions, the design provided a great deal of assurance of security. This came in two broad categories:

a. No entropy source failure could ever make the outputs look obviously bad, because a DRBG that is instantiated from a known value will still produce outputs that will pass all known statistical tests, assuming that the conditioned entropy source is still working correctly, and

b. The only critical failures possible are failures to instantiate the component DRBG of the design securely. If the component DRBG is instantiated securely, then the entropy source can immediately cease functioning, without leading to a catastrophic loss of security in any enhanced NRBG design.

Neither of these conditions can be assumed about a basic NRBG design. A Basic NRBG with no cryptographic processing of outputs has nothing with which to provide a fallback to computational security. A basic NRBG with an unapproved component DRBG may provide such a fallback, but the validation lab cannot reasonably be expected to evaluate the unapproved DRBG ~~algorithm~~ for security, and so can make no assumptions about the design's security for validation purposes. A basic NRBG will typically contain no element that provides assurance that its outputs will pass all statistical tests, though ~~common~~ designs may contain internal mixing functions that will mask entropy source failures from at least some statistical tests. Intermittent entropy source failures that would cause no problems for other designs can lead to critical failures in a basic NRBG.

A Basic NRBG will ~~be said to~~ suffer a critical failure if any k-bit output ($k<256$) is expected to be successfully guessed with less than $2^{0.9*(k-1)}$ trials[13].

Comment [ebb95]: Need to explain the numbers.

5 Constructions for Combining and Chaining RBGs

5.1 Combining RBGs

RBGs may be combined using either multiple approved RBGs, or using approve one or more approved RBGs and one or more unapproved RBGs. Combining RBGs might be used for a number of reasons, including:

13MUCH more is needed here, but we need to discuss it and hash it out first. Basic NRBGs' validation is almost entirely done at the level of the entropy source.

a. The desire to use an unapproved DRBG that is believed to be superior in security over an approved DRBG; combining the approved and unapproved DRBGs would comply with ANS X9.82.

b. The desire to combine DRBGs or NRBGs driven by different entropy sources or based on different primitives or design principles for increased assurance.

c. The desire to combine RBGs from different implementers or contained on different modules in order to obtain increased assurance.

Combining RBGs is an excellent way of meeting the requirements of this Standard for an RBG, while gaining whatever security properties are desired from some unapproved design in which the designer has enormous confidence. Existing designs that have been evaluated outside the ANS X9.82 process (designs that have been published and subjected to extensive peer review and analysis) and designs that incorporate DRBGs ~~algorithms~~ that are approved in this Standard, but which are believed by the designer to be highly secure, are all good candidates for use in a combined RBG.

The construction for combining RBGs provides assurance that the resulting combined RBG will be no weaker than the strongest component RBG, *assuming the RBGs are seeded independently*. Note, however, that there is no assurance that the combined RBG will be substantially stronger than the strongest component RBG.

A major potential pitfall to using a combined RBG is the dillution of entropy. Given k bits of total entropy available for instantiating a DRBG, using the full k bits to instantiate ~~one DRBG gives something close to k bits of practical security. Using half the entropy to~~ instantiate each of two DRBGs, and using them in a combined RBG construction, gives something close to $k/2$ bits of practical security. A combined RBG **should** be used only when the risk of dillution of entropy is outweighed by the expected gains in security. A natural approach is to instantiate one component DRBG with k bits of entropy from the source, and the other with all remaining available entropy, so that at least one DRBG is likely to get sufficient entropy even if the entropy souce is failing. In this case, an approved DRBG **shall** be substantiated with sufficient entropy for the target security level. This construction allows N component RBGs, at least one of which is approved.

Comment [ebb96]: A picture of this construction would be helpful.

5.1.1 Instantiation

Each component RBG **shall** be handled independently for instantiation and reseeding. Each RBG **should** be instantiated with a unique personalization string, although the personalization strings may be closely related, e.g., differing only in a single index byte. Each component RBG **shall** be provided with unique entropy input, not related in any way to that provided to the other component RBGs. Seed files, if used, **shall not** be shared among component RBGs.

5.1.2 Reseeding

Each component RBG may be reseeded independently ~~from time to time~~. Any entropy input used to reseed a component RBG **shall not** be reused or in any way related to those

used to reseed other component RBGs. There is no natural notion of reseeding a combined RBG, because there is no assurance that unapproved RBGs will even support such an operation or any of the security requirements associated with it.

5.1.3 Generation

To produce r bits of output from the combined RBG construction, each component RBG **shall** generate r bits of output, and the r-bit intermediate outputs **shall** be exclusive-XORed together to produce a combined r-bit final output. Additional input may be provided to any or all of the component RBGs. The additional input for each component RBG **shall** be unique, although it may be closely related, e.g., differing only by an index byte. The component RBG outputs used to generate a combined RBG output (i.e., the intermediate r-bit outputs) **shall not** be individually available.

The combined RBG outputs can support a request for information-theoretic security, for computational security up to some security level s, and prediction resistance at a security level, based on the properties of its component RBGs. In particular:

a. The combined RBG construction **shall** include at least one component RBG from this Standard.

b. The combined RBG construction **shall** be permitted to support an s bit or lower security level, if at least one approved component RBG from this Standard supports an s-bit security level. In this case, the combined RBG construction **shall** be considered equivalent to an approved DRBG with an s-bit security level.

c. The combined RBG construction **shall** be permitted to support prediction resistance for the s-bit or lower security level, if at least one ~~X9.82~~ Approved component RBG supports prediction resistance at an s-bit security level.

d. The combined RBG construction **shall** be considered equivalent to an approved NRBG, if at least one component RBG is an approved NRBG.

e. The combined RBG construction **shall** be considered equivalent to an approved Composite RBG, if at least one component RBG is an approved Composite RBG, and if the composite RBG(s) is(are) given the request for information-theoretic security whenever the combined construction generates information-theoretically secure outputs.

f. The combined RBG construction may omit the intermediate outputs from some component RBGs when computing a given combined RBG output, if the RBGs whose outputs are included allow the combined RBG construction ~~are not required for the combined RBG~~ to support whatever properties are required for the combined output.

5.2 Construction: Chaining DRBGs (Instantiating Subordinate DRBGs from a Master DRBG)

Part Three ~~of the standard~~ strongly recommends using different DRBG instantiations for different consuming applications. For example, a different DRBG instantion might be used for generating nonces than for generating AES keys. Separate instantiations provides some protection against cryptanalysis of the DRBGs, because an attack that requires a large number of known outputs might lead to the compromise of the RBG used to generate the nonce (which produces large numbers of outputs that are exposed to the world), but not to the RBG used to generate the AES key (whose outputs are never provided as plaintext). This construction may be used to derive entropy for multiple subordinate DRBGs from a single master RBG, which may be any approved DRBG.

It is important to chain the DRBGs in a sensible way in order to avoid spreading the limited entropy ~~that may be available among many RBGs~~ in a way that makes the resulting system much less robust against entropy source failures.

5.2.1 Requirements

The ~~only~~ requirements on RBGs used in this construction are as follows:

a. The subordinate RBGs **shall** be Approved DRBGs, and **shall** claim a security level that is no greater than the security level of the master RBG.

b. The subordinate RBGs **shall not** claim to provide prediction resistance.

c. The master RBG **shall not** be used for any purpose other than instantiating RBGs.

d. The master RBG **shall** provide seed material for reseeding the subordinate DRBGs, as needed.

5.2.2 Instantiation of the Master RBG

The master RBG is the only RBG with access to the live entropy source; if there are multiple entropy sources available, they **shall** be combined to support the master RBG. Instantiation is performed in accordance with for the specific RBG construction and implementation.

The master RBG **shall** not be used for any purpose other than supporting the subordinate DRBGs.

Comment [ebb97]: This is stated in Section 5.2.1.

5.2.3 Instantiation of the Subordinate DRBGs

Each subordinate DRBGs **shall** be instantiated as follows:

a. Each subordinate DRBG **should** be provided with a unique personalization string. The personalization strings may differ only by some small part of the value, such as an index byte.

b. The entropy input for each subordinate DRBG **shall** be requested from the master DRBG using a Generate request for at least the minimum number of bits needed to provide the required seed material for the subordinate DRBG. Each request **should** include a request for prediction resistance.

c. If the subordinate DRBG supports being instantiated with full entropy input, it may treat the entropy input from the master DRBG as full entropy input from a conditioned entropy source.

Comment [ebb98]: They all can support full entropy. I don't think you are saying what you mean.

5.3 Construction: Using RBGs as Entropy Sources

Entropy input for a DRBG can be acquired from an entropy source, or from an approved RBG that has access to a live entropy source. An RBG that is used to provide entropy input is called a *source RBG*; an RBG that requests entropy input from a source RBG is called a *target RBG*. Note that this construction is closely related to the construction in Section 5.2. In that construction, subordinate DRBGs can be used to compartmentalize the risks from cryptanalysis, and fresh entropy is not generally expected to be available for the subordinate DRBGs. However, in this construction, an RBG with access to a live entropy source is used to provide fresh entropy.

5.3.1 Source RBG Requirements

A source RBG **shall** be one of the following:

a. An approved NRBG,

b. An approved Composite RBG, or

c. An approved DRBG supporting prediction resistance, with a security level no lower than that of the target RBG.

Each of these RBGs ~~necessarily~~ has an entropy source available ~~to it~~ upon demand. An RBG without access to an onboard entropy source **shall not** be used to produce entropy input in this construction. A source RBG **should not** be used to produce outputs directly to a consuming application. The following requirements apply to the source RBG, based on the properties of the target RBG:

d. When the target RBG is operating as an NRBG (that is, the target RBG is either an NRBG or a Composite RBG generating full-entropy outputs), the source RBG **shall** generate entropy input for the target RBG while operating as an approved NRBG.

e. When the target RBG is operating as a DRBG with a security level of s bits, the source RBG **shall** generate entropy input for the target RBG while operating as either an approved NRBG, or as a DRBG with prediction resistance and a security level of at least s bits.

f. If the target RBG is a DRBG supporting full-entropy input, it may process the entropy input from the source RBG as full-entropy input. However, that input

shall not be assessed as having full entropy for NRBG constructions unless the source RBG is operating as an NRBG.

Comment [ebb99]: This needs to be reworded for clarity. Too much confusion about the inputs and outputs.

Annex A: Security Considerations

This section is a shambles now, but will eventually hold some of the sideline comments I noted about security issues, and some explanations about why I do some of the things I do in these constructions.

A.1 DRBG Instantiation

a. *If the entropy source fails entirely, but the personalization string is different for each time the DRBG is instantiated (e.g., the mechanism uses a timestamp), then the DRBG's outputs will show no obvious pattern of weakness, though an attacker who knows of the entropy source failure can detect the problem and predict any unseen DRBG outputs.*

b. *If the entropy source provides much less entropy than expected, say m bits, then an attacker can detect the problem (and exploit it) doing a 2^m attack for each attacked DRBG instance.*

c. *If the personalization string is unguessable to the attacker, but doesn't vary between DRBG instantiations, and the entropy source fails entirely, the attacker will notice repeating DRBG output sequences, but will have no way of knowing any bits that he has not yet observed.*

d. *If the personalization string is unguessable to the attacker, and the entropy source partially fails, so that it produces only m bits for the instantiation, then the attacker expects to have to observe outputs from about $2^{m/2}$ outputs to detect the problem, at which point he will know only the bits that he has seen from repeating DRBG output sequences. If the DRBG is never instantiated more than $2^{m/2}$ times, the attacker will never even recognize the weakness, and will be unable to exploit it.*

This has consequences for validating the entropy source: If the personalization string is unguessable to the attacker (it contains a secret with at least s bits of min-entropy), then a failure in the entopy source will not cause a critical failure unless the number of bits of min-entropy provided is less than lg(number of instantiations in DRBG lifetime)/2.

XXX Reseeding for Composite RBGs

The reseed is necessary before any computationally-secure bits are output, to protect the unconditional security of the last block of the previous output.

Note: This is one of those areas that makes me glad we're writing this up in its own part. The problem here is fairly subtle (I'm pretty sure /dev/random's design missed it.) The basic problem is that backtracking resistance never promises more than the security level. So, if we don't reseed before generating a computationally-secure output immediately after an information-theoretic one, the more powerful attacker we care about for the information-theoretic secure outputs can violate our backtracking resistance, and recover the previous internal state of the DRBG, and thus determine the information-theoretic secure output.

As an example, suppose you first generate a full-entropy output (pump 256 bits of min-entropy into AES-128-CTR, and then produce 128 bits of full-entropy output). I then request a few hundred bits of output from the *computationally secure* side. If you didn't reseed first, and I was able to do a little more than a 128-bit search, I'd end up knowing the key and counter that the computationally-secure output started with. I'd also know that new key/counter values are generated by running the old generator. So, I'd now guess the previous key, and see whether D_{guessed_key}(new_key) and D_{guessed_key}(new_counter) were only one apart. If so, I'd almost certainly have the right key. (I expect one false positive.) Now, I know the previous key and counter, so I know the information-theoretically secure output.

This stuff is tricky.

Annex B: References

References include:

Yarrow and Yarrow160

Peter's paper

Peter's Cryptlib code

PGP source code

/dev/random source code

EGD

the Truerand paper

Our DRBG cryptanalysis paper

Lisa et al's paper on security proofs for DRBGs

Intel RNG documents (specifies an integrated DRBG + entropy source....)

Declassified Clipper/Fortezza RNG details

NIST Statistical Test Suite

Diehard Statistical Test Suite

Werner's papers and standards documents, as applicable

Others?