X9.82 Part 4: Constructions for RBGs
[[Boilerplate]]

1. *Notes for X9.82 Editing Group*

- This is a really extensive rewrite; while I've reused a fair bit of text, I've gone through each of the constructions and rewritten or cut them out.
- I've cut down the number of options in many places, since there seemed little benefit in having unlimited numbers of them.
- I've made much greater use of Elaine's DRBG mechanisms, rather than trying to redo that work myself. I'm not sure where the best place to handle that stuff is (basically, the interaction between a DRBG algorithm and its entropy source), but I'm absolutely sure we don't want two incompatible versions, one here, one in part 3. This enormously simplifies things.
- I've left the external entropy source stuff in, but simplified it. I think the subset I have kept here belongs here; the rest of what I wrote before either should go away entirely, or should go to Part 2.
- I've made a lot of use of pseudocode, and I've mostly stuck with Python-inspired pseudocode because that's the language I'm most comfortable in. I'll say up front that I can't think of this stuff without putting it in object oriented terms in my own head, and that's probably coming out in the document. Thus:
    - An internally-seeded DRBG construction is an object that contains a DRBG algorithm and an entropy source, and provides a fixed interface to the outside world.
    - External accumulation, buffering, and conditioning constructions return outputs that look just like the outputs from an entropy source. When I've coded up bits of this, those things provide the same interface as an entropy source, because that way, we can use those constructions anywhere we use an entropy source. (These three constructions are just ways of hammering an approved entropy source into a new one with somewhat different shaped outputs.)

2. New Entries for Glossary
    2.1. Random Bit Generator (RBG) – the full mechanism that produces random bits for some consuming application, including any entropy sources, deterministic algorithms, and other components. [[This had better agree with Part 1!]]
    2.2. DRBG – An RBG providing only computational security.
    2.3. DRBG Algorithm – The deterministic algorithm used in a DRBG mechanism; in this Standard, Part 3 specifies the DRBG algorithms used in the DRBG mechanisms.
    2.4. DRBG Mechanism – The DRBG algorithm wrapped in a package with its entropy source and other sources of information. In this Standard, Part 3 specifies DRBG mechanisms. [[I hope this is consistent with Part 3; I think there's some lurking confusion between DRBG algorithms and mechanisms!]]
    2.5. NRBG – An RBG providing information-theoretic security. [[This had better agree with Part 1!]]

**Comment [ebb1]:** Part 1 already has a definition.

**Comment [ebb2]:** Part 1 already has a definition. This information belonfgs in the text, not the definition.

**Comment [ebb3]:** The DRBG Mechanism consists of the DRBG algorithms and the envelopes that check input parameters, acquire the entropy bits, call the appropriate DRBG algorithm and saves/retrieves the internal state. To make a complete RBG (a DRBG, in this case), the DRBG mechanism needs to be combined with the a source of entropy input.

**Comment [ebb4]:** Part 1 already has a definition. This information belongs in the text, not the definition.

2.6. Basic NRBG – An NRBG whose security guarantees are entirely dependent on its entropy source, as it does not guarantee a fallback to an approved DRBG. Note that a Basic NRBG may have cryptographic components. [[Do we define this in Part 1?]]

2.7. Enhanced NRBG – An NRBG which both promises information theoretic security, and also guarantees a fallback to an approved DRBG if the entropy source fails. [[Do we define this in Part 1? I think so, so make sure it's consistent!]]

2.8. Composite Access – Access to the DRBG mechanism of an enhanced NRBG in a secure way.

2.9. External Accumulation of Entropy – Combining of one or more outputs from an Approved entropy source into a single output that is shorter, but retains approximately the same entropy as the original outputs. Note that external accumulation is done *outside* the entropy source. [Note that a definition for Approved entropy source needs to be determined for ANSI. For the govt., it could be something like validated and certified.]

2.10. External Buffering of Entropy – Storage of one or more outputs from an Approved entropy source to support bursts of entropy requests that otherwise could not be met by the entropy source. Note that external buffering is done *outside* the entropy source.

2.11. External Conditioning of Entropy Source – Processing one or more outputs from an Approved entropy source into an output string with full entropy that is suitable for use directly as a conditioned entropy source. Note that external conditioning is done *outside* the entropy source.

2.12. Persistent State – State of an RBG that is reliably retained over long periods of time, despite disruptions such as the removal of power.

[Note: the above is preliminary information. The document begins below with the introduction. Each Section title has been styled as a header.]

**Comment [ebb5]:** Part 1 already has a definition. This information belongs in the text, not the definition.

**Comment [ebb6]:** Part 1 already has a definition. This information belongs in the text, not the definition.

**Comment [ebb7]:** What does it mean top have an Approved entropy source, when Part 2 will not be providing Approved entropy sources. For the govt., can say words like validated and certified.

**Comment [ebb8]:** This needs to be defined.

**Comment [ebb9]:** I don't think this is quite the right nuance. I think it's more the ability or condition of reliably retaining the RBG internal state.

# 1 Introduction

The preceding parts of this document have:

- 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,
- Provided guidance for developing approved *entropy sources*, mechanisms that provide truly unpredictable bits from some nondeterministic process, and
- Specified a number of *DRBG* mechanisms containing cryptographic algorithms that, when used correctly, are expected to produce bits that are indistinguishable from ideal random bits, up to the specified security level of the instantiation.

At a high level, a consuming application needs to be able to interact with an RBG. In this part of the Standard, guidance is provided for constructing RBGs from the various components from Parts 2 and 3 into RBGs that will provide output bits that meet some claimed security goal. An RBG must:

- Be powered on or instantiated,

- Generate bits – produce (pseudo) random bits according to specified security requirements, and
- Self test – perform internal testing to check that the RBG is continuing to operate as designed and implemented.

An RBG *construction* is an Approved method of building an RBG from an assortment of RBG components. In this document, constructions are explicitly specified by the heading "**Construction:**". The available components are:

- Entropy sources, as specified in Part 2; an entropy source may not provide full entropy.
- Conditioned entropy sources, as specified in Part 2; a conditioned entropy source provides full entropy.
- DRBG mechanisms, as specified in Part 3, consist of DRBG algorithms plus management code.
- Derivation functions, as specified in Part 3; functions for securely mapping a variable-length input string to a fixed-length output string.

~~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, unless otherwise explicitly stated.~~

~~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.~~

**Comment [ebb10]:** Define.

This part of the Standard is arranged as follows: Section 2 discusses RBG fundamentals—definitions, pseudocode conventions, etc. Section 3 discusses interfacing with entropy sources, providing constructions for externally manipulating entropy sources to fit specific requirements of a full RBG implementation. These constructions provide approved ways to externally accumulate entropy from a relatively sparse Approved entropy source into a much more condensed form. Constructions are also provided for conditioning entropy sources externally; note that these constructions are necessarily less efficient than conditioning routines that take into account knowledge of some underlying probability model for the nondeterministic behavior of the source. Section 4 discusses constructions for DRBGs. Section 5 discussed constructions for NRBGs. Section 6 discusses methods of using multiple RBGs together in a secure and approved way.

**Comment [ebb11]:** Personal pronouns (like we) don't belong in the Standard. Rewording has been suggested.

# 2 RBG Fundamentals

## 2.1 Definition 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, and some deterministic algorithm to generate random bits from that unguessable state (typically a DRBG algorithm). ~~[[verify this agrees with Part 1]]~~

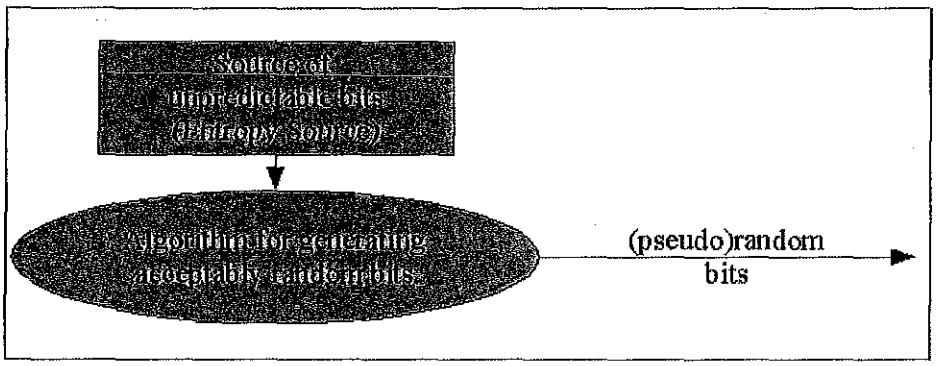



*Figure 1: High-Level Outline of an RBG*

Figure 1 provides a very high-level conceptual view of how any RBG must be designed, ~~which tracks closely with the different parts of this standard.~~ Part Two focuses on the entropy source—the component that provides unpredictable bits for the RBG. Part Three focuses on ~~some~~ algorithms for generating random bits—the components that obtain unpredictable bits from the entropy source, and use them to generate output bits that are indistinguishable from ideal random bits. Part One and this part of ANS X9.82 describe the whole RBG; Part One describes RBGs at a conceptual level, defining terms and concepts, while this part specifies how they are to be constructed from components described in Parts Two and Three. ~~Note that in some cases, an RBG is built from only a conditioned entropy source; in that case, there is no algorithm for generating the random bits outside of the entropy source; this is known as a Basic NRBG, and it is discussed in Part 2.~~

**Comment [ebb12]:** And other components that are not necessarily Part 3 DRBGs.

**Comment [ebb13]:** Not defined. Will Part 2 have this?

## 2.2 Entropy Sources

An entropy source, as discussed ~~at length~~ in Part 2 ~~of this Standard~~, is a mechanism for producing bit strings that cannot be completely predicted, and whose unpredictability can be quantified in terms of min-entropy.

Approved entropy sources may provide full entropy bitstrings, or may provide bitstrings whose entropy characteristics are known. When the Approved entropy source provides full entropy bitstrings, a conditioning function has usually been included as part of the entropy source; in this case, the term (Approved) *conditioned entropy source* is used. When the Approved entropy source does not provide full entropy bitstrings, the entropy source may not have included a conditioning function; without providing full entropy bitstrings the entropy source cannot be classified as a (Approved) conditioned entropy source. If full entropy bitstrings are required, the conditioning function may be provided external to the entropy source; the combination of the entropy source and the external

conditioning function results in a conditioned entropy source that is also approved by this Standard.

Part 4 uses Approved entropy sources in three ways: to provide entropy input for DRBG mechanisms, to provide full-entropy strings in some NRBG constructions, and to provide sufficient entropy to achieve full entropy in outputs of other NRBG constructions. Part 4 also provides constructions for ~~accessing an entropy source through a RBG~~, and for ~~externally~~ accumulating, buffering, and conditioning entropy output from an entropy source.

### 2.3 DRBGs

A DRBG (Deterministic Random Bit Generator) is an RBG that provides security only up to some computational limit, specified by a security level; that is, the DRBG provides *computational security*. For example, an AES-CTR_DRBG using a 128-bit key from Part 3 provides a 128-bit security level. The ~~promise made by this~~ DRBG ~~is that its~~ outputs will be indistinguishable from ideal random bits to an attacker who cannot do about $2^{128}$ work. This ~~means that the~~ DRBG can be used to support any application whose security level is 128 bits or less. It cannot be safely used to support an application with a higher security level~~; there may be ways to defeat the DRBG's security for an attacker who can do $2^{128}$ or more work~~.

**Comment [ebb14]:** I don't think this is needed here.

A DRBG consists of a DRBG mechanism as specified in Part 3, together with a source of entropy input. A DRBG mechanism consists of instantiate, uninstantiate, generate and test functions, and may optionally include a reseed function; the instantiate, generate and reseed functions contain DRBG algorithms to perform the required functions. A DRBG mechanism's functions may be contained within a single device, or may be distributed across multiple devices. The source of entropy input may be an NRBG, another DRBG or an entropy source whose entropy characteristics are known. A ~~full~~ DRBG must have a source of entropy input available at least once~~,~~ to instantiate it. The DRBG may or may not have entropy available at any other time.

Several terms are used in Part 4 to describe the availability of a source of entropy input to a DRBG or DRBG function. The term *reliable* is used to describe an entropy input source that is a readily available (i.e., available at all times). The term *live entropy input source* refers to an Approved NRBG or an Approved entropy source whose characteristics are known. A non-live entropy input source refers to a properly instantiated DRBG (the source DRBG) that provides entropy input for another DRBG (the target DRBG). Note that the source DRBG could actually be a chain of DRBGs; the highest DRBG in the chain **shall** obtain its entropy input from a live entropy input source, at least during instantiation (i.e., the highest DRBG in the chain shall receive its entropy input from an Approved NRBG or an Approved entropy source whose characteristics are known.

A DRBG requests entropy input using a call similar to the following:

(*status*, *entropy*, *input*) = **Get_entropy** (*k*, *n*, *max_length*),

where *k* is the amount of entropy required, *n* is the minimum length of the *entropy_input* bitstring to be returned, and *max_length* is the maximum length allowed for the *entropy_input*. While Part 3 requires that a *status* be returned from the function, the *status* will be ignored for purposes of the discussions in Part 4.

A properly instantiated DRBG remains secure even without additional entropy input as long as its internal state and entropy input are not compromised. A ~~full~~ definition of DRBGs appears in Part 1 of this Standard, while a complete specification of Approved DRBG mechanisms and requirements for their secure implementation are provided in Part 3 of this Standard.

## 2.4 NRBGs

Comment [ebb15]: Need to introduce composite access, since it's mentioned in Section 2.6.

An NRBG (Nondeterministic Random Bit Generator) is an RBG that provides output bits that are indistinguishable from ideal random bits to any observer, even one with unlimited computing power--that is, it promises *information-theoretic security*. An NRBG requires a live entropy source to be available at all times in order to operate. NRBGs are divided into *Basic NRBGs*, which simply consist of a conditioned entropy source without cryptographic protection, and *Enhanced NRBGs*, which combine a conditioned entropy source with an Approved DRBG to ensure a fallback to computational security even if the entropy source should silently fail. Two methods for constructing Enhanced NRBGs are provided:

Comment [ebb16]: Do you mean without a DRBG? It could contain a hash function.

- Using an Approved conditioned entropy source and an Approved DRBG, and XORing the outputs (see Section X), and
- Using an Approved NRBG to provide entropy input to an Approved DRBG in a manner that provides full entropy outputs from the DRBG.

## 2.5 Using RBGs Together

There are many situations in which an RBG design may use individual RBGs as components. For example, there is a construction in Section 4 for combining multiple RBGs together into a cascaded RBG, and others for using one RBG to provide entropy input for a DRBG.

## 2.6 Pseudocode and Definitions

Many constructions and algorithms are best described in pseudocode. The following notation will be used for pseudocode in the remainder of this part of the Standard.

- A DRBG algorithm is best considered as an object that supports three methods (functions). These are denoted as follows:
  - DRBG.Instantiate (seed_material)
  - DRBG.Reseed(seed_material, additional_input)
  - DRBG.Generate (bits, additional_input)

- A DRBG mechanism is an object that contains a DRBG algorithm, a call to a source of entropy input and some administrative information to allow the caller of the DRBG to ignore all details of acquiring (pseudo) random bits.
  - DRBG.Instantiate (nonce,personalization_string)
  - DRBG.Reseed()
  - DRBG.Generate(bits,additional_input,prediction_resistance_requested)
- An NRBG always supports the following two functions:
  - NRBG.Instantiate ()
  - NRBG.Generate (bits)
- An NRBG that is configured to offer composite access also supports the function:
  - NRBG.Composite_Generate (bits)
    Note that such an NRBG has a specified security level for the DRBG.
- An RBG always supports one function:
  - RBG.Generate ()
- An entropy source supports one function:
  - EntropySource.Get_entropy () -- returns (e,s), where e = entropy estimate and s = the string. When the entropy source is conditioned, e is the length of s in bits.
- x||y represents the concatenation of bitstring x with bitstring y.
- By convention, unless otherwise specified, integers are 32-bit unsigned, and when used as bitstrings, they are represented in big-endian format.

Comment [ebb17]: Not yet introduced.

Pseudocode conventions are not intended to constrain any real world implementation; the goal is to have a consistent notation to describe how the constructions in this section work. Thus, the pseudocode shows an entropy source returning an (entropy estimate, string) pair, but a real-world implementation, for example, might only return the string, with the entropy estimate always assumed to be 1.

## 2.7 Using a Derivation Function

In Part 3, two "derivation functions" are provided, of the form

$$output = \mathbf{df}(input, n),$$

where *input* is a bitstring to be condensed, and $n$ is the requested length of the output string. Note that for DRBGs in Part 4, $n$ is the minimum length of the *entropy_input* to be returned (see Section 2.3).

The derivation functions use a cryptographic mechanism (either a hash function or a block cipher) to map an input string to an arbitrary size, while distributing the unpredicability throughout the resulting output. When a construction method requires the use of a derivation function, one of these two derivation functions in Part 3 **shall** be used.

The following requirements apply to the parameters of the derivation function:

- When the block cipher derivation function is used, let $r$ be the security strength of the block cipher.

- When the derivation function uses a hash function, let *r* be the security strength of the hash function.

# 3 Interfacing with Entropy Sources

## 3.1 Overview

An entropy source is needed for any RBG. For NRBGs, entropy sources are expected to be available at all times; for DRBGs, sources of entropy input may not always be available. However, an RBG may need to use the entropy source with somewhat different parameters than it supports; for example, the entropy source may continuously provide entropy at a low rate, while the RBG occasionally requires large amounts of entropy. In this section, a number of constructions are provided for adapting the parameters of the entropy source to the requirements of the larger RBG by which it is used.

**Comment [ebb18]:** We shouldn't imply the that sources of entropy are normally available. For most RBGs currently in use, this is not the case. This has been rewritten to be neutral.

## 3.2 Construction: Entropy Accumulation

The DRBG mechanisms defined in Part 3 specify 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 implementation may not be able to support processing an enormously long string. *Accumulation* of a bitstring containing entropy is required when the entropy source is producing long output strings with sparse entropy, which must be condensed into shorter strings in order to be used; in this case, the entropy source is not performing conditioning on the raw entropy bits to obtain full entropy.

The following construction can be used to process the output of any Approved entropy source, so that its outputs are more condensed, and thus useful in different DRBGs.

### 3.2.1 Components and Requirements

The required components and values for this construction are ~~thus~~:

- An Approved entropy source.
- An Approved *derivation function* (see Section 2.7).
- A *nonce*--a value that is of fixed-length, and that is guaranteed to never repeat during the accumulation of entropy into the final *n*-bit output from the accumulate algorithm (see Section 3.2.2). The simplest implementation of this uses a large counter. The nonce **should not** repeat in the lifetime of the RBG.
- An *accumulator*--a value of *n* bits that is used to accumulate up to *k* bits of entropy at a time. Note that an *n* bit accumulator can accumulate up to *k/2* bits of entropy in this construction.
- A *counter*--a value that records the amount of entropy that has been accumulated.

**Comment [ebb19]:** Again, what is meant by an Approved entropy source? This needs to be discussed at X9F1.

Let *k* be the amount of entropy to be accumulated. The value of *k* has the following restrictions when this construction is used:

- $k$ **shall** be less than or equal to the security strength $r$ of the cryptographic algorithm used in the derivation function (i.e., $k \leq r$), and
- $2k$ **shall** be less than or equal to the requested length of the output string $n$ (i.e., $2k \leq n$).

### 3.2.2 Description of the ~~External~~ Accumulation Algorithm

In order to condense a bitstring containing $k$ bits of entropy down to an $n$-bit string, a temporary buffer (*accumulator*) is used; this buffer is initially set to the all-zero string. As many requests as necessary are made to the entropy source to get at least $k$ bits of entropy. For each output returned from the entropy source, a nonce is prepended, and the derivation function is applied to the resulting string~~, requesting $n$ bits of output~~. The output of the derivation function is XORed into the temporary buffer.

Note that the nonce need only be unique for each input string processed in producing one $n$-bit final output with at least $k$ bits of entropy.

### 3.2.3 Pseudocode Description of the ~~External~~ Accumulation Algorithm

~~We can describe this algorithm in pseudocode:~~

```
# Accumulate k bits of entropy in a buffer of n bits.
def accumulate_entropy(EntropySource, k, n):
    if 2*k>n: raise an error condition and exit

    counter = 0
    accumulator = 0...0.
    while counter<k:
        # e is entropy estimate for this string, x is the string.
        e,x = EntropySource.Get_entropy()
        nonce = MakeNextNonce()
        accumulator = accumulator XOR df(nonce||x,n)
        counter = counter + e

    return k,accumulator
```

The output from this algorithm may be used exactly like any other entropy source query. This algorithm can be used to build a kind of wrapper around any approved entropy source, to provide more condensed outputs for the convenience of the other components of an RBG.

**Comment [ebb20]:** I suggest that we rename the Get_entropy function in Part 3 to Get_entropy_input so that Parts 3 and 4 do not define functions with the same name. The name Get_entropy is more appropriate in Part 4.

**Comment [ebb21]:** $k$ need not be returned. When changing to non-python pseudocode, the status and accumulator will need to be returned (if using the Get_entropy call).

**Comment [ebb22]:** Since there's no 3.2.3.2, then this should be eliminated.

#### ~~3.2.3.1 Accumulating Entropy in a Single DF Call~~

Note that this construction can be used ~~in a very simple way~~ to condense $k$ bits of entropy from a long input string to an $n$-bit string by simply calling the derivation function once. That is, the above pseudocode could be condensed to:

```
e,s = EntropySource.Get_entropy()
if e<k: raise an error condition and exit
```

*return df(0||s, n)*

This pseudocode fragment would condense a single value from the entropy source to an $n$ bit result.

Comment [ebb23]: Does this really add anything? Do we still have the restriction that 2k <= n, or is this something else?

### 3.3 Construction: External Conditioning of Entropy Source Output

Many constructions in this part of the Standard require conditioned entropy sources. Similarly, implemented DRBGs may need to obtain entropy input from conditioned entropy sources.

This *External Conditioning* construction allows an RBG design to externally process the output of an Approved entropy source that may not have an internal conditioning capability in order to obtain a conditioned entropy source with full entropy.

This construction can be used to process the output of any Approved entropy source, so that its outputs are conditioned, and may be used anywhere a conditioned entropy source's outputs may be used.

Note that when *entropy_input* is obtained for a DRBG using this construction, the requested entropy $k$ (see Section 2.3) equals the minimum length $n$ of the requested *entropy_input*.

[Note: The figure was not readable. Also need a reference to the figure if it's to be present.]

#### 3.3.1 Components and Requirements

The required components and values for this construction are ~~thus~~:

- An Approved entropy source.
- An Approved *derivation function* (see Section 2.7).
- A *nonce*--a value that is of fixed-length, and that is guaranteed to never repeat during the conditioning of entropy into the final $n$-bit output. The simplest implementation of this uses a large counter. The nonce **shall not** repeat during the computation of a single conditioned output, and **should not** repeat during the lifetime of the lifetime of the RBG.
- The nonce **should not** repeat in the lifetime of the RBG.
- An *accumulator*--a value of $n$ bits that is used to accumulate up to $n$ bits of entropy at a time. Note that at least $2n$ bits of entropy are (supposedly) obtained in the accumulator during this process, but only $n$ bits of entropy are returned as output when using this construction; The over-sampling is provided in order to allow for inaccurate entropy estimates.
- A *counter*--a value that records the amount of entropy that has been accumulated.

Comment [ebb24]: The DRBG instantiation, for example. Do we needs a definition for RBG lifetime?

Let $n$ be the amount of entropy to be accumulated and the length of the output from this algorithm. $n$ **shall** be less than or equal to the security strength $r$ of the cryptographic algorithm used in the derivation function (i.e., $n \leq r$).

### 3.3.2 Text Description of the Conditioning Algorithm

In order to produce a conditioned (full-entropy) $n$ bit buffer from any Approved entropy source, a temporary buffer (*accumulator*) is used; this buffer is initially set to the all-zero string. As many requests as necessary are made to the entropy source to get at least $2n$ bits of entropy. For each output returned from the entropy source, a nonce is prepended, and the derivation function is applied to the resulting string. The output of the derivation function is XORed into the temporary buffer. The final result in the buffer contains an $n$-bit conditioned entropy source output.

Note that the nonce need only be unique for each input string processed in producing one $n$-bit final output with $n$ bits of entropy.

### 3.3.3 Pseudocode Description of Conditioning Algorithm

~~We can describe this algorithm in pseudocode:~~

```
# Produce a single n-bit conditioned entropy source output.
def condition_entropy(EntropySource, n):

    counter = 0
    accumulator = 0...0.
    while counter<2*n:
        # e is entropy estimate for this string, x is the string.
        e,x = EntropySource.Get_entropy()
        nonce = MakeNextNonce()
        accumulator = accumulator XOR df(nonce||x,n)
        counter = counter + e

    return n, accumulator
```

**Comment [ebb25]:** Return status instead of n. n is an input.

The resulting bits may be used in exactly the same way as the outputs of a conditioned entropy source as specified in Part 2. ~~This construction allows a wrapper to be put around any Approved entropy source, which results in a conditioned entropy source.~~ Thus, any Approved entropy source may be used with this construction to support a Basic NRBG and the Enhanced NRBG XOR Construction, which are described in Sections [redacted] and [redacted].

#### ~~3.3.3.1 Conditioning from a Single Entropy Source Output~~

Note that this construction can be used ~~in a very simple way~~ to condition a long entropy source output. The following pseudocode describes ~~a way to do this~~ conditioning using a single call.

```
e,s = EntropySource.Get_entropy()
if e<k: raise an error condition and exit
return df(0||s, n)
```

This pseudocode fragment would condense a single value from the entropy source to an $n$ bit result.

**Comment [ebb26]:** Does this really add anything?

### 3.4 Construction: ~~External~~ Buffering of Entropy Source Outputs

An RBG may require entropy at a very different rate than the entropy source produces it. Within some limits, an RBG designer can use ~~external~~ buffering to allow the entropy source to ~~still~~ service the RBG. Outputs from this construction ~~given here~~ can be used in exactly the same way as outputs obtained directly from an entropy source, and this construction may be used to "wrap" the outputs of an Approved entropy source for use by other components of an RBG.

#### 3.4.1 Components and Requirements

This construction requires the following components:

- A queue capable of storing up to $n$ entropy source outputs, along with their entropy estimates. The queue ~~is an object which~~ stores a set of outputs in order, and retrieves them ~~as outputs~~ upon request in the same order. In this case, the queue's entries consist of pairs of values: an entropy estimate, and a string containing entropy.
  - **Q** represents the queue.
  - **Q**.put(*estimate*, *string*) represents inserting an (*estimate*, *string*) pair into the queue.
  - (*estimate*, *string*)=**Q**.get() is used to retrieve an (*estimate*,*string*) pair.
- An Approved entropy source, **ES**.
- A sum of the current amount of entropy contained in the queue, *sum*.

Specific requirements for this construction include:

- Each entry in the queue **shall** be irretrieveably deleted as soon as it is used.
- The ~~external~~ accumulation and ~~external~~ conditioning constructions in Sections 3.2 and 3.3 may be used to store entropy from the entropy source more densely than it is produced by the original entropy source. (This follows from the fact that the outputs from those constructions may be used anywhere the outputs from an Approved entropy source may be used.)

#### 3.4.2 Text Description of the Buffering Algorithm

~~The algorithm is very simple.~~ Whenever it is convenient, an output is requested from the entropy source, ~~and~~ stored in the queue, and the sum of the entropy in the queue is updated to include the amount of newly acquired entropy. When a request for some amount of entropy is presented ~~to the buffer~~, the requested amount of entropy is returned if there is enough entropy in the queue to satisfy the request~~, it is returned~~; otherwise, the request fails.

#### 3.4.3 Pseudocode Description of the ~~External~~ Buffering Algorithm

```
def put_entropy_in():
        e,s = ES.Get_entropy()
        Q.put(e,s)
        sum = sum + e
def get_entropy_out(n):
        if sum < n: raise an error condition and exit
```

```
tmp = ""
entropy = 0
while entropy < n:
        e,s = Q.get()
        sum = sum – e
        entropy = entropy + e
        tmp = tmp || s
return sum, tmp
```

Comment [ebb27]: This should probably be a status code (i.e., Success). Is there any reason that the calling function needs the sum? I suppose that one reason would be to look elsewhere (e.g., another entropy source) if there isn't enough entropy available at a given time.

# 4 Constructing a DRBG

## 4.1 Overview

A DRBG is an RBG that provides computational security up to a security level that is dependent on its design and the amount of entropy provided during instantiation. The entropy may be provided from a live source of entropy input during instantiation only, or the live source of entropy input may be available at all times (i.e., the entropy input source is reliable). Alternatively, another DRBG may be used as a "non-live" source of entropy input.

A DRBG with a source of entropy input only during instantiation cannot provide prediction resistance or reseeding. A DRBG with a reliable live source of entropy input (see Section X) can be instantiated and can handle requests for prediction resistance and automatic reseeding, as can a DRBG that uses another DRBG as a source of entropy input (see Section Y).

Comment [ebb28]: Is this true or not. I couldn't quite determine this from the previous wording.

## 4.2 Construction: DRBG Without a Live Source of Entropy Input

### 4.2.1 Overview

A DRBG that requires entropy input, but is without a live source of entropy input, is dependent on some properly instantiated DRBG (a source DRBG *S*) for its entropy input. The source DRBG may be used by the subordinate DRBG *T* to provide all its entropy input or to provide entropy input in addition to what is obtained from a live entropy input source (e.g., to provide entropy input at those times when the live entropy input source is not available, or to obtain additional assurance in case there is a failure by the live entropy input source).

In those cases where a live entropy input source is not available, DRBG *T* cannot support prediction resistance, and can only instantiate and reseed when it is provided entropy input from the source DRBG *S*. If DRBG *T* requires reseeding, the DRBG must suspend activity until entropy input is available.

Comment [ebb29]: Is this correct? If it can reseed using the source DRBG, ccouldn't it also provide prediction resistance, or is this relying too much on the source DRBG; or does it deopend on whether the source DRBG provides prediction resistance?

The following DRBG constructions ~~given here~~ result in (full) RBGs, i.e., RBGs that can be used, once properly instantiated, to generate random bits for cryptographic and other applications.

### 4.2.2 Source DRBG

Comment [ebb30]: This term is used instead of a seedfile, as it seems to be what is intended.

#### 4.2.2.1 Overview

A source DRBG that provides entropy input to other DRBGs that may not have access to a live source of entropy input **shall** be a properly instantiated DRBG with sufficient entropy to support its subordinate DRBGs; the security level of the source DRBG **shall** be equal to or greater than the security level of any subordinate DRBG. Any entropy acquired by a subordinate DRBG may also be provided to the source DRBG to supplement its entropy. This concept is used by several non-Approved DRBGs and is commonly known as a seedfile.

#### 4.2.2.2 Components

~~A seedfile has one component: the *SEEDFILE DRBG*.~~

Constructions that use the source DRBG will use the following notation:

- source.DRBG.instantiate(entropy_input,seed_matersial)
- source.DRBG.save(*S*) – save the entropy in *S* into the source DRBG
- *S* = seedfile.get(n,optional_additional_input) – get n bits of seed material from the seedfile, incorporating an optional additional input into the seedfile in the process

Comment [ebb31]: I don't think we need to invent calls that are already in Part 3; this should be just the Part 3 instantiate function call.

Comment [ebb32]: This should be just the generate function request in Part 3.

#### 4.2.2.3 Instantiating a Source DRBG

The source DRBG (or highest DRBG in the source DRBG chain) **shall** be instantiated from an NRBG or an entropy source whose characteristics are known with sufficient entropy to support the subordinate DRBGs (see Section [redacted] and Part 3). A personalization string **shall** be used.

#### 4.2.2.4 Providing New Entropy to the Source DRBG

One common operation with a source DRBG is to provide ~~some~~ new entropy to the source DRBG that has been introduced into a subordinate. This will improve the chances that source DRBG output provided to other subordinate DRBGs will be sufficiently independent of the output provided to previous subordinate DRBGs. ~~have sufficient entropy, even in the face of silently failing or unavailable entropy sources.~~

Comment [ebb33]: Is this correct?

Let *S* be a string containing ~~some~~ entropy. To save this to the source DRBG, the following DRBG operation is performed:

```
def source.DRBG.save(S):
    tmp = source.DRBG.Generate(state_handle, 8,requested_security_strength,
        prediction_resistance_request, S)
```

The returned byte (returned as *tmp*) is discarded. The result of this operation is that the entropy from *S* is stored in the internal state of the source DRBG, and will be available to subordinate DRBGs in the future.

#### 4.2.2.5 Generating Entropy Input for a Subordinate DRBG from a Source DRBG

The other common operation performed by the source DRBG is to generate entropy input for a subordinate DRBG. This may be the only source of entropy input for a subordinate DRBG, or the source DRBG's output may be combined with ~~some~~ output from an entropy source or other RBG output for added assurance. A new entropy input with $k$ bits of entropy is generated as follows:

*def Get_entropy_input (k, k, k):*
*return source.DRBG.Generate(state_handle, k, k, prediction_resistance_request, additional_input)*

Comment [ebb34]: The Get_entropy_input request in Part 3 does not currently support providing additional input to the entropy input source. The instantiate function does have a personalization string, and the reseed function does have additional input, but there is currently no provision for passing them on. Do we need it?

Comment [ebb35]: Can't this just be a reference to the Part 3 generate function call?

### 4.2.3 Construction: Subordinate DRBG With Persistant Memory

#### 4.2.3.1 General Discussion

A subordinate DRBG with or without a live source of entropy input and with the capability to maintain its internal state over time may use the source DRBG to obtain entropy input during instantiation. There are a number of complications introduced by the lack of a live source of entropy input. For example:

- Instantiation and reseeding are possible only when a source DRBG is available to provide entropy input.
- If the subordinate DRBG requires reseeding (e.g., the reseed interval has been reached; see Part 3), the subordinate DRBG suspends activity.

#### 4.2.3.2 Components and Requirements

The construction requires two components:

- *T*, the subordinate DRBG
- Entropy input for from the source DRBG for instantiation.

Some requirements:

- The source DRBG **shall** be capable of supporting the intended security level of the subordinate DRBG; that is, the security level of the source DRBG **shall** meet or exceed the intended security level of the subordinate DRBG.
- The entropy input provided by the source DRBG for instantiation **shall** be protected from compromise.

#### 4.2.3.3 Instantiation

The subordinate DRBG is instantiated as specified in Section 4.2.2.5.

#### 4.2.3.4 Generation

Generating outputs is done by the subordinate DRBG as specified in Part 3. If entropy is available in any additional_input that is provided during generation, the subordinate DRBG **should** accumulate this entropy.

Comment [ebb36]: shall?

#### 4.2.3.5 Reseeding

Reseeding the subordinate DRBG is possible only when entropy input is available with sufficient entropy. If the source DRBG is available to provide entropy input, the entropy input provided by the source DRBG and any additional input provided by the consuming application is used to reseed the subordinate DRBG instantiation as specified in Part 3.

### 4.2.4 Construction: Subordinate DRBG Without Persistent Memory

A subordinate DRBG without persistent memory does not have the ability to maintain its internal state over a long period of time, although short-term memory may be available. A new instantiation needs to be created frequently, and may, in fact, need to be created whenever the subordinate DRBG is required to generate pseudorandom bits. A source DRBG may be used to provide the required entropy input to create a new instantiation for the subordinate DRBG. An example of this case is a DRBG on a smart card that has no persistent memory. Whenever the smart card is inserted into a reader, the DRBG on the smart card (the subordinate DRBG) is instantiated by acquiring entropy input from a properly instantiated DRBG (the source DRBG) that is built into the reader. The subordinate DRBG is used to supply random bits to the application using the smart card.

At a high level, this construction works as follows:

1. A source DRBG is instantiated from some entropy source (possibly another RBG) when it is available, such as during manufacturing or device setup.
2. At some point in time (e.g., during power up of the device containing the subordinate DRBG), the subordinate DRBG is instantiated. This is accomplished by requesting entropy input from the source DRBG (see Section 4.2.2.5) and using that entropy input to instantiate the subordinate DRBG. Any additional input that is available from the application using the subordinate DRBG should be provided in the personalization string during subordinate DRBG's instantiation.
3. During operation, application data that might contain ~~some~~ entropy is saved~~, and periodically used as additional input in one-byte generate requests to the source DRBG (see step 4). The resulting one-byte outputs are discarded.~~
4. At a later point in time (e.g., during power down of the device containing the subordinate DRBG), the subordinate DRBG generates a $k$ bit output, where $k$ is [redacted]. This output is used as additional input, along with any other available application data that was acquired in step 3, in a ~~one-byte~~ generate request to the source DRBG (see Section 4.2.2.4). ~~The one-byte output is discarded.~~
5. If the application rarely or never has a power down, then a $k$-bit value from the subordinate DRBG is periodically (e.g., once a day) ~~be~~ generated and used as additional input in a ~~one-byte~~ generate request to the source DRBG (see Section 4.2.2.4).

Comment [ebb37]: This needs to be reworded.

#### 4.2.4.1 Components and Requirements

This construction consists of two components and two values:

- A source DRBG as described in Section 4.2.2.
- A subordinate DRBG, a DRBG that is periodically instantiated, and that is used to generate random bits for applications.

- Entropy input furnished for at least the first instantiation.
- *length*, the number of bits needed to instantiate the subordinate DRBG.

The two-stage external DRBG supports the following operations:

- ~~FirstInstantiate(seed_material)—A first instantiation, which gets the seedfile to a secure starting point.~~
- ~~Instantiate(optional_input)—Instantiation of the ephemeral DRBG~~
- Generate(additional_input)—Generation of outputs from the ephemeral DRBG
- Reseed(seed_material)—Reseeding when external seed material is made available
- Save(optional_input)—Saving the state of the ephemeral DRBG back into the seedfile, along with some optional input
- ~~A real implementation probably also includes a way to shut down CURRENT, but this isn't important for the construction.~~

Comment [ebb38]: Shouldn't define anything that is also defined in Part 3.

#### 4.2.4.2 Source DRBG Instantiation

The source DRBG is instantiated only once, from either another RBG or an entropy source as specified in Section 4.2.2.3.

#### 4.2.4.3 Instantiating the Subordinate DRBG

The consuming application instantiates the subordinate DRBG when random outputs are required, and the subordinate DRBG is not instantiated (e.g., when power has not been available to maintain the internal state for any previous instantiation). This is accomplished by requesting the source DRBG to generate output, using that output as entropy input to instantiate the subordinate DRBG. If the consuming application has available entropy, it can be provided in the optional additional input of the generate request to the source DRBG (see Part 3).

[Pseudocode may indeed be helpful here, but it needs to be constructed to use the Part 3 generate and instantiate functions.]

#### 4.2.4.4 Providing Entropy from the Subordinate DRBG to the Source DRBG

Over time, the subordinate DRBG may accumulate a small amount of entropy from the sequence of requests it processes, and ~~a large amount of~~ entropy from any additional inputs that are provided by the subordinate DRBG's consuming application. Periodically (e.g., during power down of the subordinate DRBG), the subordinate DRBG can provide this entropy ~~back~~ to the source DRBG as specified in Section 4.2.2.4.

#### 4.2.4.5 Reseed of the source DRBG

When entropy input is available, the source DRBG may be reseeded. If the subordinate DRBG is currently using the source DRBG, its internal state is saved, along with the new entropy input. Otherwise, the new entropy input is saved directly to the source DRBG. Reseding the source DRBG is discussed in Section 4.2.3.5.

Comment [ebb39]: Which internal state is saved and why?

~~*TwoStageDRBG.Reseed(seed_material):*~~
~~*if CURRENT is active:*~~
~~*TwoStageDRBG.Save(seed_material)*~~

~~else:~~

~~SEEDFILE.Save(seed_material)~~

In either case, if the source DRBG has been compromised, but the new entropy input for the source DRBG is unknown, the new internal state of the source DRBG (and also the internal state of the subordinate DRBG, if it's being used) is also unknown to the attacker. On the other hand, if the source DRBG has not been compromised, even new entropy input chosen by the attacker cannot lead to a compromise of the source or subordinate DRBGs.

## 4.3 Constructing a DRBG With a Live Source of Entropy Input

### 4.3.1 General Discussion

The ideal situation for a ~~full~~ DRBG (the target DRBG *T*) is to have reliable live source of entropy input (e.g., an Approved conditioned entropy source or an Approved NRBG). In addition to entropy input from a live entropy input source, DRBG *T* may obtain entropy input from another (source) DRBG as discussed in Section 4.2.2.

The live entropy input source provides bit strings with a claimed amount of entropy. An example of an entropy input source would be a ring oscillator that is sampled one hundred times per second, and for which extensive analysis has been performed to determine that each sampled sequence of 100 bits ~~sampled~~ has at least 80 bits of entropy.

~~Any DRBG with access to a live entropy input source can access that source: if the source DRBG claims *k* bits of security, then each new request for *k* or more bits of output with prediction resistance from the source DRBG can be assumed to contain *k* bits of min-entropy.~~

When DRBG *T* has a reliable live entropy input source, instantiation and reseeding ~~and instantiation~~ can be done on demand, requests for prediction resistance can be honored, and when DRBG *T* reaches the end of its seed period (see Part 3), the DRBG can reseed itself. If the DRBG claims *k* bits of security, each new request for output with prediction resistance can be assumed to provide *k* bits of min-entropy.

~~An internally seeded DRBG may use a seedfile, as described below, but does not require one.~~

~~Note that in this section, we use a DRBG Mechanism, not a DRBG Algorithm. This means that the DRBG is expected to handle its own reseeding, prediction resistance requests, etc.~~ Two constructions are provided ~~for internally seeded DRBGs~~, one using only a live source of entropy input, and the other augmenting the live entropy input source with the output of a source DRBG.

### 4.3.2 Construction: DRBG With a Live Entropy Input Source

This construction ~~simply~~ uses the DRBG mechanisms specified in Part 3~~, and is probably the simplest construction in part 4~~.

~~4.2.2.1 Components and Requirements~~

~~A DRBG that uses only a live entropy input source consists of one component:~~

- ~~**DRBGMECH** – A DRBG mechanism with a readily available entropy input source.~~

~~4.4.2.2 Instantiation, Reseeding, and Generation~~

Essentially, all requests to the construction amount to requests to the DRBG mechanism as specified in Part 3.

~~*def InternallySeededDRBG.instantiate(nonce,personalization_string):*~~
~~*DRBGMECH.Instantiate(nonce,personalization_string)*~~

~~*def*~~
~~*InternallySeededDRBG.generate(bits,additional_input,prediction_resistance_requested):*~~
~~*DRBGMECH.generate(bits,additional_input,prediction_resistance_requested)*~~

~~*def InternallySeededDRBG.reseed(additional_input):*~~
~~*DRBGMECH.reseed(additional_input)*~~

### 4.3.3 Construction: DRBG With a Live Entropy Input Source and a Source DRBG

A DRBG ***T*** with reliable access to an entropy input source can be augmented with the output of a source DRBG *S*. This provides a certain level of insurance against silent failure of the entropy input source. There are a number of details to get right in using the output of the source DRBG to ensure that the full benefit is received.

#### 4.3.3.1 Components and Requirements

This construction consists of two components and one parameter:

- A DRBG mechanism as described in Part 3
- A source DRBG as described in Section 4.2.2.
- *length*, the number of bits needed to instantiate DRBG ***T*** from the source DRBG. This is typically 3/2 times the security level of the DRBG mechanism.

#### 4.3.3.2 First Instantiation

Both the target and source DRBGs are instantiated using separate entropy inputs from the live entropy input source. The target DRBG is instantiated as discussed in Section 4.3.2. The instantiation of the source DRBG is discussed in Section 4.2.2.3. Subsequently, the DRBG construction may be used to generate outputs.

**Comment [ebb40]:** This was my understanding.

#### 4.3.3.3 Subsequent Instantiations of the Target DRBG

Each time after the first that the target DRBG is instantiated, an entropy input is included from the source DRBG. This is used as part of the personalization string. Immediately after instantiation, an output from the target DRBG is used to update the internal state of the source DRBG.

```
def target.DRBG.instantiate(nonce,personalization_string):
    if personalization_string is not present:
        personalization_string = ""
    S = SEEDFILE.Get(length)
    DRBGMECH.instantiate(nonce,S+personalization_string)
    SEEDFILE.Save(DRBGMECH.generate(length))
```

Comment [ebb41]: Needs to be fixed.

At the end of this process, if the live entropy input source ~~in the DRBGMECH~~ is working properly, both the target DRBG and the source DRBG are "secure" (i.e., xxxx). If the entropy input source has silently failed, but the source DRBG has been properly instantiated, then both the target DRBG and the source DRBG as secure.

Comment [ebb42]: To use the term "secure state", we'll need a definition. Also, "state" as used in X9.82 refers to an internal state.

#### 4.3.3.4 Reseeding the Target DRBG

Each time that the target DRBG is reseeded, the source DRBG contributes information for the reseeding, and benefits from any entropy provided for reseeding.

```
def ISDSeedfile.reseed(additional_input):
    if additional_input is not present:
        additional_input = ""
    S = SEEDFILE.Get(length)
    DRBGMECH.reseed(S+additional_input)
    SEEDFILE.Save(DRBGMECH.generate(length))
```

Comment [ebb43]: To be fixed.

As before, security of each component cannot be made worse by this operation, but can be made better.

#### 4.3.3.5 Generation of Random Bits by the Target DRBG

The generation of random bits by the target DRBG uses the live entropy input source and output from the source DRBG only when prediction resistance is requested (see Part 3). When the target DRBG requests *n* bits of output with prediction resistance, the entropy input **shall** be the XOR of *n* bits obtained from the live entropy input source with *n* bits obtained from the source DRBG.

Comment [ebb44]: Can output from the source DRBG be used here?

# 5 Constructing an NRBG

## 5.1 Overview

An NRBG is a mechanism for producing bits with information theoretic security (equivalently, full entropy). These bits are expected to be indistinguishable from ideal random bits to any attacker, no matter how computationally powerful. There are three constructions for NRBGs:

- Basic NRBG—An NRBG based solely upon a conditioned entropy source
- Enhanced NRBG: Xor Construction—An NRBG based upon combining the outputs of a conditioned entropy source with those of any DRBG construction. (The DRBG could in principle be an externally seeded one, but it's hard to imagine this being done in practice.)

- Enhanced NRBG: Oversampling Construction—An NRBG based upon using an internally seeded DRBG construction in a mode which provides information-theoretic security.

## 5.2 Constructing a Basic NRBG

### 5.2.1 General Discussion

The simplest possible NRBG is a basic NRBG; it uses only a conditioned entropy source, and relies for its security only upon the properties of that source.

Unlike the other constructions for NRBGs, a Basic NRBG can fail in an obvious and disasterous way if its entropy source misbehaves even in relatively small ways. Compared with internally seeded DRBGs and enhanced NRBGs, basic NRBGs are more vulnerable to failure because:

- 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.
- If the underlying DRBG algorithm of one of these other constructions is instantiated securely, then the entropy source can immediately cease functioning without leading to a catastrophic loss of security in any enhanced NRBG design.

In all other construction in part 4, an entropy source that deviates just slightly from correct behavior leads to a very small security impact; the DRBG algorithms mask any misbehavior, and the practical impact is a small decrease in the expected work to guess the DRBG's working state.

### 5.2.2 Components

The only component of a basic NRBG is the conditioned entropy source, **CES**.

### 5.2.3 Instantiation

No instantiation process is necessary for a basic NRBG, except for whatever process initializes **CES**.

### 5.2.4 Generation

A request for $n$ bits of output from the basic NRBG is fulfilled by a process like the following:

```
BasicNRBG.Generate(n):
    tmp = ""
    sum = 0
    while sum<n:
        # Note: e = len(s) by definition!
        e,s = CES.Get_entropy()
        tmp = tmp || s
```

*sum = sum + e*
*return leftmost n bits of tmp*

### 5.2.5 Additional Concerns

The fundamental problem with a basic NRBG is that there is no fallback in case of some undetected failure of the entropy source. At present, basic NRBGs shall not be approved. In the future, we expect extensive requirements on the underlying conditioned entropy source, in terms of:

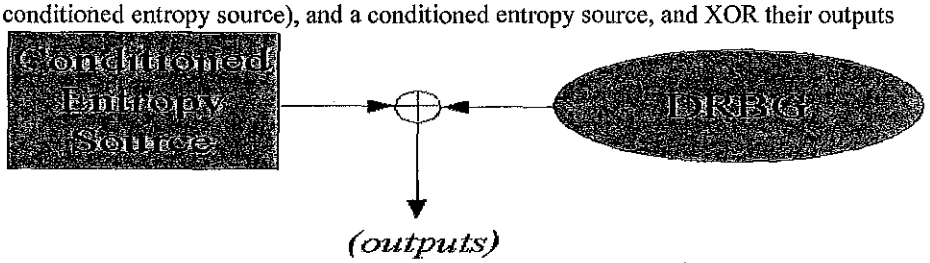
- Strenuous design validation and testing
- Comprehensive continuous testing
- Use of approved cryptographic mechanisms to condition the entropy source, such as the external conditioning construction specified above.

[[Is there something more to be said here at this point?]]

## 5.3 Construction: Enhanced NRBG XOR Construction

### 5.3.1 General Discussion

The XOR construction is an extremely simple way to construct an enhanced NRBG from two components: a DRBG and a conditioned entropy source. Conceptually, we take a DRBG mechanism (potentially with its own entropy source, though it might draw on the conditioned entropy source), and a conditioned entropy source, and XOR their outputs




together to provide information theoretically secure random values. The security argument for this is straightforward: if the conditioned entropy source is functioning correctly, the outputs of the DRBG are being XORed with full-entropy, truly random bits, and so the results will also be truly random. On the other hand, if there is some kind of failure in the conditioned entropy source, the DRBG outputs will mask that failure to any attacker who cannot do enough work to defeat the DRBG's security level.

### 5.3.2 Components

The two components are:

- **CES**, a conditioned entropy source
- **D**, a DRBG construction which may:
  - Use **CES** or the underlying entropy source of **CES** as its entropy source.
  - Have an independent entropy source
  - Be externally seeded, and draw its seed material from **CES**.

[[I need to expand this to two constructions, so I can show how to safely use an externally-seeded DRBG construction with this NRBG construction.]]

### 5.3.3 Instantiation

To instantiate the NRBG, the following process is done:

```
def Xor_NRBG.instantiate():
    D.Instantiate()
```

Optionally:

- **D** can be reseeded one or more times to gain assurance.
- **CES** can be used to generate an additional inputs for the reseeding call.

### 5.3.4 Generation

To generate an *n* bit output, the conditioned entropy source is used to produce *n* bits, the DRBG is used to produce *n* bits, and the two bitstrings are XORed together. The result is returned. In pseudocode, this is

[[Verify all the pseudocode calling conventions!]]

```
def Xor_NRBG.generate(n):
    tmp = ""
    sum = 0
    while sum<n:
        # Note: e = len(s) by definition!
        e,s = CES.Get_entropy()
        tmp = tmp || s
        sum = sum + e
    tmp = leftmost n bits of tmp
    tmp = tmp XOR D.Generate(n)
    return tmp
```

Note: the call to **D**.Generate() may request prediction resistance, if this is available.

### 5.3.5 Composite NRBG Access

This NRBG construction can also support a request for DRBG outputs, as might be needed for higher-volume requests than the conditioned entropy source could fulfull. This is done by simply asking the component DRBG, **D**, to fulfill the generate request.

```
def XorNRBG.CompositeGenerate(n):
    return D.Generate(n)
```

## 5.4 Construction: Enhanced NRBG Unicity Distance Construction

### 5.4.1 General Discussion

A second construction for an enhanced NRBG is called the unicity-distance construction. This does not need a conditioned entropy source, and operates in a somewhat different way. The NRBG consists of a DRBG mechanism, which contains an available entropy

source and supports prediction resistance. Outputs are generated by the DRBG, with at least twice as much entropy being fed into the DRBG as bits generated by it. Assuming a minimally good DRBG algorithm, this leads to full-entropy outputs.

[This Figure is not displayed, and needs to be referenced, anyway.]

### 5.4.2 Components

The NRBG has one component, and a related parameter:

- **D**, a DRBG mechanism with an internal entropy source and prediction resistance support.
- *k* is the security level supported by **D**

### 5.4.3 Instantiation

Instantiating the NRBG requires simply instantiating the DRBG:

```
def UnicityNRBG.Instantiate():
    D.Instantiate()
```

### 5.4.4 Generation

Generating output from the DRBG is based on making many requests for outputs with prediction resistance, with the knowledge that the DRBG has been reseeded with at least *k* bits of additional entropy for each request.

```
def UnicityNRBG.Generate(n):
    tmp = ""
    sum = 0
    while sum<n:
        s = D.Get_entropy(k/2, prediction_resistance_requested=1)
        tmp = tmp || s
        sum = sum + k/2
    return leftmost n bits of tmp
```

### 5.4.5 Composite NRBG Access

Again, the underlying DRBG can be accessed safely, so long as it is accessed with prediction resistance requested.

```
def UnicityNRBG.CompositeGenerate(bits):
    return D.Generate(n,prediction_resistance_requested=1)
```

# 6 Using RBGs Together

## 6.1 Overview

An RBG may be constructed from a number of components, which may themselves be RBGs, or may be DRBG or NRBG mechanisms. This section discusses how these RBGs **shall** be composed ~~to accomplish a number of different goals~~.

**Comment [ebb45]:** Are there NRBG mechanisms?

## 6.2 Construction: Combining RBGs into a Single Cascaded RBG

### 6.2.1 General Discussion

A cascaded RBG may be constructed using either multiple Approved RBGs, or using one or more Approved RBGs and one or more unapproved RBGs. Constructing a cascaded RBG might be done for a number of reasons, including:

- The desire to use an unapproved DRBG that is believed to be superior in security over an Approved DRBG; combining the Approved and unapproved DRBGs would comply with this Standard.
- The desire to combine DRBGs or NRBGs that are driven by different entropy sources or based on different primitives or design principles for increased assurance.
- The desire to combine RBGs from different implementers or contained on different modules in order to obtain increased assurance.

Comment [ebb46]: Note the indentaiton.

Designing and implementing a cascaded RBG 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 (e.g., designs that have been published and subjected to extensive peer review and analysis) and designs that incorporate DRBGs that are not 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.

A properly constructed cascaded RBG provides assurance that the resulting RBG will be no weaker than the strongest component RBG, assuming that the entropy input for any DRBG in the cascaded RBG is independent of the entropy input for any other DRBG in the RBG. Note, however, that there is no assurance that the cascaded RBG will be substantially stronger than the strongest component RBG. Each RBG in this construction is assumed to be self-contained; that is, each RBG is expected to be able to support the instantiation of the DRBG and the generation of bits from that RBG.

### 6.2.2 Overview of the XOR Construction for a Cascaded RBG

This construction allows *N* component RBGs, at least one of which is Approved, to be combined to make a cascaded RBG that conforms to this Standard.

A special case of this cascaded RBG is discussed in Section 5 as one of the two approved constructions for enhanced NRBGs.

The security level and properties of the cascaded RBG are ~~determined as follows~~:

- The cascaded RBG ~~construction~~ **shall** include at least one component RBG from this Standard.

- The security level of the cascaded RBG is ~~given a claimed security level~~ equal to the highest security level of any Approved RBG from which the cascaded RBG is composed. Note that if one of its Approved component RBGs is an NRBG, then the cascaded RBG can support any security level, including the infinite security level whenever the NRBG is functioning correctly. In this case, output from the cascaded RBG may be used in exactly the same way as any Approved NRBG's outputs.

Comment [ebb47]: Does this term appear anywhere else in X9.82?

- The cascaded RBG is capable of supporting prediction resistance if either:
  - One of its Approved component RBGs is an NRBG
  - One of its Approved component RBGs with the same security level as the cascaded RBG supports prediction resistance (i.e., if the cascaded RBG supports a 256-bit security level, then the cascaded RBG can support prediction resistance only if one or more of the Approved component RBGs has a security level of 256 bits).

### 6.2.3 Preliminaries: Interfacing with Component RBGs

For the pseudocode below, ~~we adopt~~ the following convention is used: If an implemented component RBG cannot support some parameter (such as *additional_input* or *prediction_resistance_requested*), then that parameter is omitted from the function call. Thus, if a given RBG, *R*, does not support the *additional_input* parameter in its Generate function, the pseudocode of

R.Generate(..., *additional_input*)

**shall** be taken to mean

R.Generate(...), where "..." is used to represent other parameters that are used.

### 6.2.4 Instantiation of DRBGs in a Cascaded RBG

Each DRBG in a cascaded RBG **shall** be instantiated prior to using the RBG to generate bits. Instantiation of the DRBGs can be summarized by the following pseudocode:

*Cascaded_Instantiate([requested_instantiation_security_strength], [prediction_resistance_flag], [personalization_string]):*

*For each Approved component DRBG, R:*
*~~nonce = MakeNextNonce()~~*
*R.Instantiate(requested_security_strength, prediction_resistance_flag, personalization_string)*
*For each unapproved component DRBG, R:*
*R.Instantiate(whatever input parameters are required)*

The following requirements apply to the instantiation of DRBGs for ~~the XOR construction of~~ cascaded RBGs:

- Each component DRBG **shall** be provided with unique entropy input that is not related in any way to that provided to the other component DRBGs.
- Source DRBG (instantiations), if used, **shall not** be shared among component RBGs.

- Each DRBG **shall** use a distinct personalization string; personalization strings may be made distinct by the inclusion of a counter or other non-repeating value.

### 6.2.5 Reseeding of DRBGs in a Cascaded RBG

The component DRBGs within a cascaded RBG may be reseeded ~~independently~~ at any time by a consuming application, as follows:

*Cascaded_Reseed(additional_input):*
*For each Approved component DRBG R: R.Reseed({state_handle for R}, additional_input)*
*For each unapproved component DRBG R with a reseed capability: R.Reseed().*

The following requirements apply to the cascaded DRBG reseed:

- Each component DRBG that supports a reseed() capability **shall** be given a reseed request.
- The entropy input used to reseed one component DRBG **shall** be independent of the entropy input used to reseed another component DRBG.
- Each Approved component DRBG that supports a reseed capability and accepts additional input **shall** use any additional input provided.

Note that a component DRBG may also control its own reseeding (i.e., a DRBG may reseed itself whenever required).

### 6.2.6 Generation Of Bits from a Cascaded RBG

The cascaded RBG generate call is as follows:

*Cascaded_Generate(N,requested_security_strength, full_entropy, prediction_resistance_requested additional_input):*

*if prediction_resistance_requested and the cascaded RBG cannot support prediction resistance, raise an error condition and exit.*
*if full_entropy is requested and the cascaded RBG cannot support full entropy, raise an error condition and exit.*
*tmp = N bit block of binary zeros.*
*For each component DRBG, R:*
*if R is an approved DRBG:*
*tmp = tmp XOR R.Generate( state_handle, N, requested_security_strength, prediction_resistance_flag, additional_input, prediction_resistance_requested)*
*else: tmp = tmp XOR R.Generate(N)*
*For each component NRBG: R: tmp = tmp XOR R.Generate(N)*
*return tmp*

**Comment [ebb48]:** Need to rewrite this to use the full_entropy version, when required.

The following requirements apply to the combined RBG generate function:

- The *additional_input* **shall** be provided to all DRBGs that support additional input.
- The *prediction_resistance_request* **shall** be provided to all DRBGs that support prediction resistance.
- No intermediate values for *tmp* or outputs of individual RBGs used to generate the output from the cascaded RBG **shall** be released.

### 6.3 Obtaining Entropy from an RBG

Consider a source RBG ***S*** with a reliable entropy source. A DRBG (a target DRBG ***T***) can request outputs from ***S*** with either full entropy or prediction resistance, ~~to get access to the underlying entropy source~~. The source RBG may be either an Approved DRBG with a reliable entropy input source or an Approved NRBG.

**Comment [ebb49]:** This would not be a direct access, since the outputs from the entropy source would be transformed in some way by S.

#### 6.3.1 Construction: Obtaining Entropy from an NRBG

When *n* bits of entropy are required from an NRBG *S*, the calling DRBG (*T*) can simply request *n* bits of output from the NRBG. These output bits may be treated exactly like the output from a conditioned entropy source. Thus, in pseudocode, a call like:

*e,s = EntropySource.Get_entropy()*

to a conditioned entropy source could be rewritten as:

*e = k*
*s = NRBG.Generate(k)*

where *k* is the amount of entropy that is requested..

#### 6.3.2 Construction: A DRBG Obtains Entropy Input from a DRBG with a Reliable Entropy Source

When *n* bits of entropy are required from a DRBG (the source DRBG *S*) that provides *k* bits of security, and that DRBG has a reliable entropy source, the calling DRBG (*T*) accumulates the *n* bits from the source DRBG *S* using as many calls as are needed. The source DRBG is given a request to generate a maximum of *k* bits with prediction resistance in each call, and the result is assumed to have a min-entropy equal to the number of bits obtained in the call. Each output may be treated as if it is the output from an Approved entropy source.

Pseudocode like the following would be used to get *n* bits of entropy from an Approved entropy source:

*sum = 0*
*tmp = the null string*
*while sum<n:*
  *e,s = EntropySource.Get_entropy()*

*tmp = tmp || s*
*sum = sum + e*
*return the leftmost n bits of sum*

This could be ~~re~~written as follows using any source DRBG with prediction resistance and a *k*-bit security level:

*sum = 0*
*tmp = the null string*
*k = security_level*
*while sum<n:*
*tmp = tmp || DRBG.Generate(state_handle, k, k, prediction_resistance_requested, [additional_input])*
*sum = sum + k*
*return the leftmost n bits of sum*

## 6.4 Construction: Using a DRBG to Provide Entropy Input to Other DRBGs

Comment [ebb50]: If the Target DRBG obtains its entropy from an NRBG, the call is different. May need to handle this separately. Note that this section is essentially the same as Section 4.2.2.

A DRBG may obtain its entropy input from another properly instantiated Approved DRBG. Note that this construction is very similar to the previous construction; the difference is that this construction uses an Approved DRBG to provide entropy input~~, without the assumption that the entropy input has any independent entropy~~.

Comment [ebb51]: Don't understand this.

[The figure can't be seen. Also, need a reference to it.]

The target DRBG ***T*** may obtain its entropy input for instantiation and reseeding from the source DRBG ***S*** using the Get_entropy_input call in the instantiate and reseed functions (see Part 3). The Get_entropy_input call of Part 3 results in the following invocation:

*Source.Generate(state_handle, this k+k/2 requested_security_strength,prediction_resistance_request, additional_input )).*

Comment [ebb52]: Thois seems to be assuming that the nonce is a random number.

where *k* is the security level. Note that (*k*+*k*/2) provides a (*k*/2)-bit random nonce (see Part 3).

The following requirements apply to instantiating a target DRBG from a source DRBG:

- The target DRBG's security level **shall** be less than or equal to the source DRBG's security level.
- The target DRBG **shall not** support prediction resistance or full entropy.
- In order to obtain entropy input for a target DRBG with a security level of *k* bits, the source DRBG **shall** generate an output of at least *k*+*k*/2 bits. This output **shall** be used only for instantiating the target DRBG, and **shall** be discarded immediately afterward.

~~The construction for a seedfile from a persistent DRBG, described below, is a special case of this construction.~~

### ~~6.4.1 Construction: Providing Entropy Input for Many DRBGs from One Properly Instantiated DRBG~~

The source DRBG may be used to provide entropy input for many target DRBGs (i.e., different DRBG instantiations), each of which may support different applications. Different instantiations may be used to compartmentalize the applications of the DRBGs, so that cryptanalysis of one DRBG instantiation does not compromise other parts of applications.

Additional requirements are:

- The source DRBG ~~mechanism~~ **shall** be instantiated with all available entropy, including nonce and personalization string ~~and/or nonce~~.
- The entropy input for each target DRBG **shall** be obtained from a distinct generate request for at least $k+k/2$ bits of output from the source DRBG, where $k$ is the intended security level of the target DRBG.
- The most exposed target DRBG **should** be the last to be instantiated, while the target DRBG whose outputs are considered the most critical to guard **should** be instantiated first.

Comment [ebb53]: This needs clarification.

Comment [ebb54]: This needs more information.

~~For example, if a sequence of ten DRBG instantiations is required, the entropy input for each would be obtained from the source DRBG *S* as follows:~~

```
drbg_list = []
for i = 1 to 10:
    drbg_list[i] = DRBG.Instantiate(S.Generate(k+k/2))
```

## Appendices

### Appendix A Security Considerations

7.1.1 Oversampling and Conditioning

### Appendix B Example Source Code for Some Constructions in Python

[[This is untested, I used it to clarify my thinking on various constructions]]

```
#####
# Python implementation of constructions section.
#
# This code is a contribution of the federal government
# and is not subject to copyright.
#
# John Kelsey, NIST, October 2005
#
# Note: This is untested demonstration code.  Don't use
# it in a production environment without testing.  (We'd love a debugged
```

```
# version ourselves!)
#####


#####
# Entropy Source Section: Three Constructions
#
# a. EntropySourceAccumulator
# b. EntropySourceConditioner
# c. EntropySourceBuffer
#####

#####
# This is the abstract base class for an entropy source. Assumptions:
#
# a. A real entropy source object must respond to get_entropy() with
# an integer,string pair, where the integer tells us the entropy estimate
# and the string carries the entropy. We get no promises whatsoever from
# the entropy source about the distribution in that string, unless the
# entropy source is conditioned, which we can tell by calling the conditioned()
# method.
#
# b. We assume that this won't always return 0 as an entropy estimate.
# If it does, we will get stuck in an infinite loop in various places.
#####
class EntropySource(object):
  # Abstract base class
  def __init__(self):
    pass
  def get_entropy(self):
    # Always return estimate,sample
    # estimate is an integer
    # sample is a string
    pass
  # Return True or 1 for conditioned entropy sources only!
  def conditioned(self):
    pass

#####
# Implementation-specific utility routine!
#####
def xorString(s1,s2):
  xl = [chr(ord(s1[i])^ord(s2[i])) for i in range(len(s1))]
  return "".join(xl)

def ExternalEntropyAccumulator(EntropySource):
```

```
    def _make_next_nonce(self):
        self.C += 1
        return "%08x"%self.C

    def __init__(self,entropy_source,df,entropy,short,long):
        self.source = entropy_source
        self.df = df
        self.k = entropy
        self.short = short
        self.long = long
        self.C = 0

        if long<2*entropy: raise ValueError,"Can't condense that far down!"

    def internally_seeded(self): return 1

    def get_entropy(self):
            counter = 0
        while counter<k:
            # e is entropy estimate for this string, x is the string.
            e,x = self.source.get_entropy()
            nonce = self._make_next_nonce()
            accumulator = xorString(accumulator,df(n,nonce+x))
            counter = counter + e

        return k,accumulator


class ExternalEntropyConditioner(ExternalEntropyAccumulator):
    def __init__(self,entropy_source,df,entropy):
        self.source = entropy_source
        self.df = df
        self.entropy= entropy
        if entropy%8<>0: raise ValueError,"Not implemented!"
        if entropy<64: raise ValueError,"Output too small to get good statistics!"
        self.C = 0

    def get_entropy(self):
            counter = 0
        while counter<self.entropy*2:
            # e is entropy estimate for this string, x is the string.
            e,x = self.source.get_entropy()
            nonce = self._make_next_nonce()
            accumulator = xorString(accumulator,df(n,nonce+x))
            counter = counter + e
```

```
        return k,accumulator[:self.entropy*8]

class InsufficientEntropy(Exception): pass

class ExternalBuffering(EntropySource):
    def __init__(self,source,queue):
        self.queue = queue
        self.source = source
        self.sum = 0

    def collect(self):
        e,s = self.source.get_entropy()
        self.queue.put((e,s))
        self.sum = self.sum + s

    def get(self,n):
        if self.sum<n:
            raise Exception(InsufficientEntropy)
        else:
            tmp = ""
            count = 0
            while count<n:
                e,s = self.queue.get()
                count = count + e
                tmp = tmp + s
            return tmp

#####
# DRBG SECTION
# Four constructions
# a. Internally Seeded DRBG -- basically the DRBG Mechanism Elaine has
#    specified.
# b. Seedfile -- A DRBG used as an entropy store.
# c. One-Stage Externally Seeded DRBG -- A DRBG that only gets instantiated
#    once, and is reseeded only when there's entropy available.
# d. Two-Stage Externally Seeded DRBG.
#####

# This is the abstract base class for a DRBG algorithm.
# Assumptions:
#  a. The DRBG algorithm object knows how to instantiate, generate, and
#     reseed according to the interface here.
#  b. The DRBG algorithm knows its own reseed requirements. When the
#     DRBG algorithm needs to be reseeded in order to continue, we raise
#     a special-purpose exception, ReseedRequired.
#  c. I have not even considered fractional byte requests here. I assume that
```

```
#       whatever conventions exist for handling these, they'll be handled
#       by the real DRBG_Algorithm implementation.
#   d.  The DRBG algorithm is responsible for failing when it is called in the
#       wrong order: it raises the NotReady exception.
#####
class ReseedRequired(Exception): pass
class NotReady(Exception): pass

class DRBG_Algorithm(object):
    # Abstract base class
    def __init__(self):
        self.security_level = None
        self.short = None
        self.long = None
        self.ready = 0
        pass
    def instantiate(self,seed_material):
        self.ready = 1

    def generate(self,bits,additional_input=None):
        if not self.ready: raise NotReady
    def reseed(self,seed_material,additional_input=None):
        if not self.ready: raise NotReady
    def uninstantiate(self):
        self.ready = 0
    def security_level(self): return self.security_level
    def parameters(self): return (self.security_level*3/2,2**32)



#####
# The internally seeded DRBG is self-contained; once it's instantiated,
# you don't really have to mess with it much.  If you demand prediction
# resistance and it doesn't support it, it will fail with a NotSupported
# exception.
#####
class NotSupported(Exception): pass

class Internally_Seeded_DRBG(object):
    def _make_seed(self,entropy,short,long):
        tmp = ""
        sum = 0
        while sum<target:
            e,s = self.entropy_source.get_entropy()
            tmp = tmp + s
            sum = sum + e
```

```
    if len(tmp)*8>long or len(tmp)<short:
        tmp = self.df(long,tmp)

    return tmp

def __init__(self,entropy_source,drbg_algorithm,df,
             nonce=None,personalization_string=None,
             pres_supported=1):
    self.entropy_source = entropy_source
    self.drbg = drbg_algorithm
    self.df = df
    self.pres_supported = pres_supported

    ##### Instantiate the DRBG:

    # Get parameters from the DRBG algorithm object
    self.short,self.long = self.drbg.parameters()
    self.k = self.drbg.security_level()

    # Decide if we need extra entropy to cover nonce.
    if nonce==None:
        nonce = ""
        target = self.k*3/2
    else:
        target = self.k

    tmp = self._make_seed(target,self.short,self.long)
    self.drbg.instantiate(tmp)

def generate(self,bits,additional_input=None,pres_req=0):

    if pres_req==1:
        if self.pres_supported:
            self.reseed(additional_input)
        else: raise NotSupported

    try:
        tmp = self.drbg.generate(bits,additional_input)
        return tmp
    except ReseedRequired:
        self.reseed(additional_input)
        tmp = self.drbg.generate(bits,additional_input)
        return tmp

def reseed(self,additional_input):
```

```
        tmp = self._make_seed(self.k,self.short,self.long)
        self.drbg.reseed(tmp,additional_input)

    def security_level(self): return self.drbg.security_level

# This is a very different kind of object.  The internally seeded DRBG takes
# care of everything internally; this one must pass all exceptions for reseed
# required back out to the caller, must require an input parameter for its
# seed material when an instantiation or reseed is required, etc.
class ExternallySeededDRBG(object):
    def __init__(self,drbg_algorithm,seed_material):
        self.drbg = drbg_algorithm
        self.drbg.instantiate(seed_material)
    def generate(self,n,additional_input=None):
        self.drbg.generate(n,additional_input)
    def reseed(self,seed_material,additional_input=None):
        self.drbg.reseed(seed_material,additional_input)
    def security_level(self): return self.drbg.security_level

#####
# The seedfile supports a very simple interface; its goal is to be a
# persistent store of entropy for a DRBG which is actually used.
#####
class Seedfile(object):
    def __init__(self,drbg_algorithm):
        self.drbg = drbg_algorithm
        self.instantiated = 0
    def instantiate(self,seed_material):
        self.drbg.instantiate(seed_material)
    def save(self,data):
        self.drbg.generate(8,data)
    def get(self,n,additional_input=None):
        return self.drbg.generate(n,additional_input)

class TwoStageExternallySeededDRBG(ExternallySeededDRBG):
    def __init__(self,seedfile,drbg_algorithm,seed_material):
        self.seedfile = seedfile
        self.current = drbg_algorithm
        self.length,x = drbg_algorithm.parameters()
        if seedfile.security_level()<drbg_algorithm.security_level:
            raise ValueError,"Seedfile can't support claimed secuirty level!"

        self.seedfile.instantiate(seed_material)
        self.ready = 0
```

```
  def instantiate(self,additional_input=None):
    seed = self.seedfile.get(self.length,additional_input)
    self.drbg.instantiate(seed)
    self.ready = 1

  def generate(self,bits,additional_input=None,pred_req=0):
    if not self.ready: raise NotReady
    if pred_req: raise NotSupported
    return self.drbg.generate(bits,additional_input)

  def save(self,additional_input=None):
    if additional_input<>None: self.seedfile.save(additional_input)
    if self.ready: self.seedfile.save(self.drbg.generate(self.length))

  def reseed(self,seed_material):
    if not self.ready: raise NotReady
    self.drbg.reseed(seed_material)
    self.save()

  def uninstantiate(self):
    self.ready = 0
    # A real implementation would zeroize DRBG!

#####
# NRBG Section
#
# Two Constructions:
# a. XorNRBG
# b. UnicityNRBG
#####

class NRBG(object):
  def generate(self,n):
    pass
  def composite_generate(self,n):
    pass

# No mechanism to guarantee frequent reseeds here; we need to make
# clear that it's acceptable for the DRBG mechanism to reseed as often
# as it likes, so long as entropy is available!
class XorNRBG(object):
  def __init__(self,drbg_mechanism,CES):
    if not CES.conditioned(): raise ValueError,"Need conditioned entropy source!"
    self.CES = CES
    self.drbgmech = drbg_mechanism
  def generate(self,n):
```

```
        if n%8<>0: raise ValueError,"Fractional bytes not implemented!"
        tmp = ""
        count = 0
        while count<n:
            e,s = self.CES.get_entropy()
            tmp = tmp + s
            count = count + e
        tmp = tmp[:n*8]
        tmp = xorString(tmp,self.drbgmech.generate(n))
        return tmp
    def composite_generate(self,n):
        return self.drbgmech.generate(n)

# This construction is trivial once we have an internally seeded DRBG
# mechanism to play with....
class UnicityNRBG(object):
    def __init__(self,drbg_mechanism):
        if not self.drbgmech.internally_seeded():
            raise ValueError,"Need internally seeded DRBG for this construction!"
        self.drbgmech = drbg_mechanism
        self.increment = self.drbgmech.security_level/2
    def generate(self,n):
        tmp = ""
        count = 0
        while len(tmp)<n:
            x = self.drbgmech.generate(n,pres_req=1)
            tmp = tmp + x[:self.increment*8]
            count = count + self.increment
        return tmp[:n*8]
    def composite_generate(self,n):
        return self.drbgmech.generate(n,pres_req=1)
```

[...]