Part 4: Constructions for Building and Validating RBG Mechanisms

[[Need to add boilerplate, glossary, references, abstract/summary/whatever. May need to add a section at end on cryptanalysis of these designs, to motivate some choices. -- JMK]]

0 New Definitions for Glossary

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

b. *DRBG Mechanism* -- An RBG mechanism providing only computationally-secure outputs.

c. *NRBG Mechanism* -- An RBG mechanism providing only information-theoretically secure outputs.

d. *Composite Mechanism* -- An RBG mechanism providing both computationally-secure and information-theoretically secure outputs, as required by consuming applications.

e. *critical failure* -- A failure of an entropy source that leads to a major loss of security, defined rather arbitrarily here as a loss of at least 20% of the claimed bits of security for a computationally secure mechanism. Basic NRBGs (which do not have any security level claimed) are arbitrarily assigned a target security level of 128 bits for purposes of determining whether some potential flaw in the entropy source leads to a critical failure

f. *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.

g. *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.

h. *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.

1 Preliminaries

This is Part 4 of the X9.82 Random Bit Generation standard. In this document, we specify how the components and definitions in Parts 1, 2, and 3 of this Standard shall be combined to provide working *RBG mechanisms*--practical mechanisms for producing random bits with the required security properties.

In the preceding sections, we have :

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

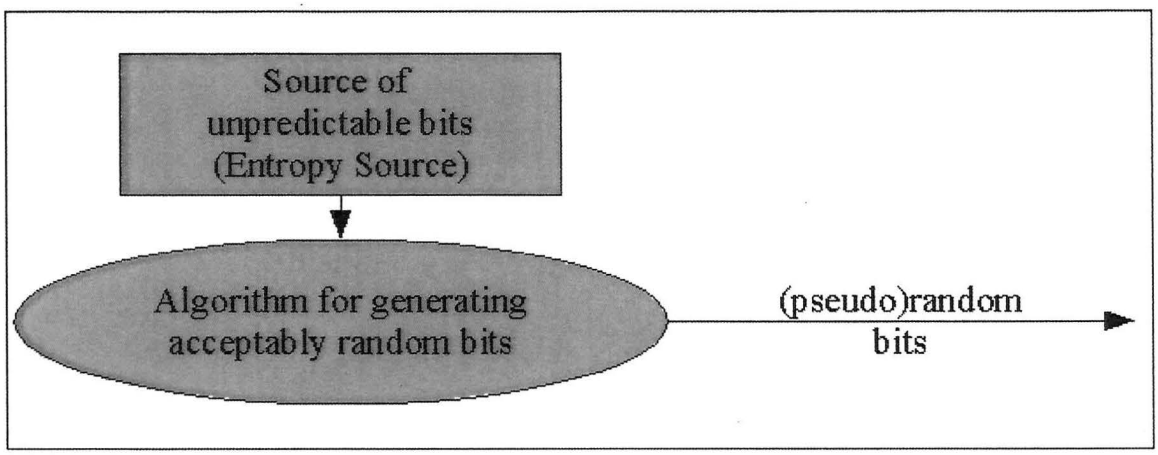
b. Developed guidance for developing approved *entropy sources*, mechanisms which provide truly unpredictable values from some nondeterministic process.

c. Specified a number of *DRBG* algorithms, cryptographic algorithms which, used correctly, are expected to produce bits indistinguishable from ideal random bits up to their specified security levels.

In this part of the standard, we provide guidance for developing, implementing, and validating RBG mechanisms up to their claimed (possibly unlimited) security level. In support of this, we also provide a number of *constructions*--ways to put together components to achieve some security goal. Much of our discussion of RBG mechanisms is informative, though there are normative aspects[[kill the weasel words somehow!]]. Constructions are normative; they specify the approved ways to do things. For example, there are two enhanced NRBG constructions permitted in X9.82; while we recognize that there are a great many equally secure ways to build an enhanced NRBG, there are even more apparently-reasonable ways to build an enhanced NRBG that turn out not to be secure. Rather than leave the reader to evaluate a design on his own, we provide two constructions and tell him to use one. Similarly, we provide a construction for using a DRBG algorithm and some mechanism for accumulating entropy to support an entropy pool design, similar in spirit to that of /dev/random or EGD. Again, we recognize that many other constructions are secure, but we have limited resources for evaluation, and so we simply provide one such construction.

1.1 Structure of an RBG Mechanism

An RBG mechanism produces random bits for some consuming application, providing some guarantees 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 mechanism 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 mechanism 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.



1.2 Entropy Management: How Entropy Sources Go Wrong

The biggest problem in building a working, secure RBG mechanism is managing entropy. There are many reasons for this: algorithms work the same way on every platform and implementation, but entropy sources are *always* dependent on implementation details, so that 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 which is not detected by the RBG mechanism, and which causes a substantial loss in security. A great deal of the practical work in building a secure RBG mechanism amounts to minimizing the probability of a critical failure by providing some level of fallback security in the design.

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

An RBG mechanism claims some security level from the list of {112,128,192,256}, or claims to be information-theoretically secure, and may also claim the ability to provide prediction resistance. For most RBG mechanisms, 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.) Naturally, if these algorithms are broken, the output bits from them are no longer suitable for some uses. The DRBG algorithms appearing in Part 3 have survived review both from within the X9.82 working group [[correct term?]] and externally, but the state of the art in cryptography does not permit a guarantee that these algorithms are secure.

For DRBG mechanisms claiming only computational security, the only requirement on

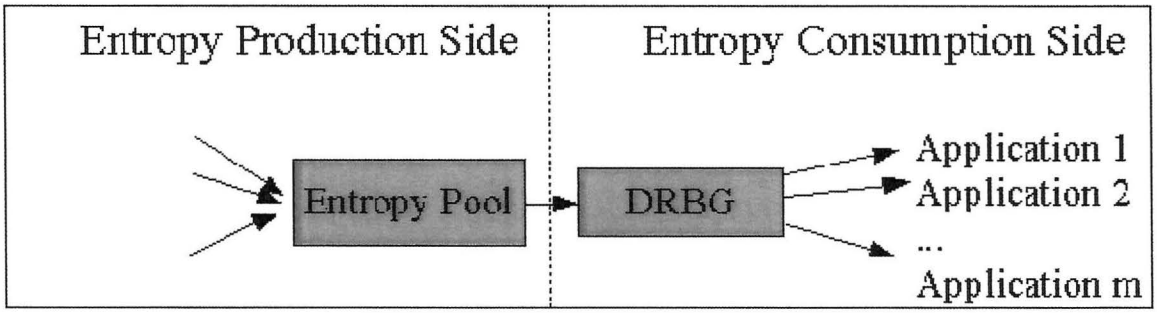
the underlying DRBG algorithm is that it support at least the same security level as the DRBG mechanism claims. For NRBG mechanisms which claim information-theoretic security, and for composite mechanisms which sometimes claim information-theoretic security, it is often surprisingly difficult to get more than the design strength of security from a DRBG algorithm. For this reason, we provide constructions to accomplish these goals in secure ways; these constructions are normative, and there are no other methods acceptable to accomplish these goals.

1.4 Relationships between Entropy Rate and Output Rate

For computationally-secure RBG mechanisms (DRBG mechanisms), or composite mechanisms that can support computationally-secure requests for random bits, the only required relationship between rate of entropy production from the entropy source and rate of outputs is that the DRBG can't *start* producing outputs until it has been properly instantiated, and it can't support *prediction resistance* requests without reseeding.

For information-theoretic secure RBG mechanism (NRBG mechanisms), the rate of production of entropy from the source must be at least the rate of output requested.

In general, software-sources (especially those based on human interaction with the computer) tend to have wildly varying rates of entropy production--a machine with a human typing or playing a video game may have a great deal of entropy available from its interactions with the user. Similarly, sources based on network activity produce more estimated entropy during times of heavy local network traffic, and sources based on system loading produce more estimated entropy when more processes or threads are running on the machine. These varying rates of entropy production make buffering (discussed below) an important system design consideration. Also, software entropy sources are typically combined together, as the software designer cannot always be certain what devices or other resources will even be available to the program on all machines.



Hardware sources, by contrast, tend to have pretty consistent performance over time. That is, 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 the rate of consumption of entropy is bursty, 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 that discard some bits, such as Von Neumann unbiasing.

1.5 A Roadmap

Part 4 is arranged as follows: First, we discuss the issues raised in implementing, using, and validating an entropy source as part of an RBG mechanism. For reasons that will become clear below, an entropy source must be validated as part of an RBG mechanism. Next, we discuss various options for building RBG mechanisms that support only computationally-bounded security, specifying in each case the requirements and optional features for how the RBG may be constructed. Finally, we discuss RBG mechanisms providing information-theoretic security, specify the allowable constructions, and explore how these may be tested and validated.

2 Entropy Sources in RBG Mechanisms

Part 2 describes entropy sources by themselves. Here, we discuss how to integrate an entropy source into a larger RBG mechanism. The focus here is on taking an existing, good entropy source design, and fitting it into the RBG mechanism. This includes:

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

b. Combining entropy source outputs together in a pool, and estimating min-entropy based on conservative strategies.

c. Validating the entropy sources for use in the RBG mechanism, 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--arbitrary defined as a loss of 20% or more of the claimed security level for computationally-secure mechanisms.)

2.1 Preliminaries

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

An entropy source has some support for testing. Conceptually, the tests are part of the entropy source (they are generally going to be tailored to a specific probability model), although they may well be implemented separately. An entropy source SHALL provide startup and periodic tests tailored to its expected behavior, probability model, and known or suspected failure modes. An entropy source MAY additionally provide continuous testing on its outputs or internal behavior, to quickly catch failures. Again, this continuous testing MAY be implemented outside the actual entropy source, but it is

conceptually part of the source, as it is designed with an intimate knowledge of the internals of the entropy source.

Testing an entropy source allows some failures to be caught, before they lead to some major loss of security in the RBG mechanism that relies upon it. They are used to decrease the probability of a *critical failure* of the entropy source--one in which the security of the RBG mechanism would be undermined.

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

The DRBG algorithms 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. When an entropy source produces entropy input which, in order to have the required amount of entropy, is simply too long for the DRBG algorithm implementation to process, either the entropy source itself or some external component must somehow condense the entropy input to a much shorter string, in an efficient way that doesn't lose very much of the input entropy. Similarly, when the RBG uses entropy in bursts, buffering can allow a relatively slow entropy source to keep up, rather than forcing outputs (and consuming applications) to wait.

2.2.1 Using the Derivation Functions

In practice, the DRBG algorithm uses the 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 do this externally, and then to feed the result into the DRBGs 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 call to instantiate or reseed the DRBG, even when the DRBG then uses the derivation function a second time to process that input.)

2.2.1.1 Using Hash_df

A hash function is a natural tool for accumulating entropy, and hash_df provides a reasonable way to use the 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 be at least the number of bits needed for instantiation. The output length SHOULD be a multiple of the hash output size for efficiency.

Recall from Part 3 that hash_df() is defined as:

hash_df(inputString, *bits*):

```
tmp = ""
ctr = 0
while bit_length(tmp)<bits:
        tmp = tmp || hash(ctr||bits||inputString)
return most significant bits bits from tmp
```

So long as the output length is known before processing begins, this can be computed on the fly, without buffering the whole input string. The output can be buffered using any of the techniques described below, or can be used directly.

2.2.1.2 Using bc_df

Recall from Part 3 that bc_df() is defined as: [[will eventually be defined as....]]

```
bc_df(inputString,bits):
        ##### Construct a padded string with length prepended.
        L = bit_length(inputString)
        S = encode(L/8) || encode(bits/8) || inputString || 0x80
        while bit_length(S)<BLOCKLEN:
                S = S || 0x00

        ##### Compute the seed to generate the output value.
        tmp = ""
        i = 0
        K = 0x010203...1f  # Truncated to KEYLEN
        while bit_len(tmp)<KEYLEN+BLOCKLEN:
                IV =  encode(i)||0x000000...00 # Truncated to BLOCKLEN
                tmp = tmp || CBC-MAC(K,IV||S)

        ##### Compute output of requested length.
        K = most significant KEYLEN bits of tmp
        X = next BLOCKLEN bits of tmp
        tmp = ""
        while bit_length(tmp)<bits:
                X = Encrypt(K,X)
                tmp = tmp || X
        return most significant bits bits of tmp
```

With bc_df, it is necessary to specify the input and output lengths before the first block of 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 they produce per bit of output; for those entropy sources, the bc_df may be a bad accumulation routine. Note that it is acceptable to zero-pad input strings. The bc_df could thus be started with the maximum length of input that might be required, but when sufficient entropy was reached in the input, the remainder of the input could be filled in with zeros.

The outputs from the bc_df can be buffered using any of the schemes described below.

2.2.3 Accumulating Entropy in a CRC

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 is no influence of the specific feedback polynomial on the entropy source's behavior. This is true because if I told you I was going to use a CRC to accumulate my entropy, and let you choose my input set, but you didn't know my feedback polynomial, then you couldn't choose messages that collided with a lot higher than expected probability. At least, this is my understanding--am I somehow missing something? --JMK]] If this is done, it SHALL be done 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. [[Otherwise, leading zeros have no effect!--JMK]]

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.

2.2.3.1 Buffering CRC Outputs

Buffering CRC outputs SHOULD use the hash-buffer construction discussed below, but MAY use any of the three buffering techniques described below.

2.2.4 A Software Entropy Accumulation Mechanism

[[I need to explain some *fast* way to do this in software, as well as in hardware. CRCs aren't all that fast in software, alas. I'm thinking of the /dev/random pool mixing function--stuff that's being done inside an interrupt routine, and so can't go out and compute a hash or some such thing. Maybe I should ask permission to use this, and just do the analysis on that design. But all the obvious designs I can see are seriously ad-hoc. Maybe we could use some universal hashing scheme? Or CRCs? Or *can* we just tell people to use a CRC if they need speed?

Is there some nice way to do a universal hash based scheme here? Or to prove something about the /dev/random pool design, or something similar? Is there an efficient way to do a big CRC in software I'm not aware of?

Doug Whiting says with a simple feedback polynomial (small number of terms), I can probably do this in 2 ops/byte in assembly on a modern Intel CPU. So maybe this is worth considering. I think the analysis of CRCs is much nicer than for these ad hoc designs....

--JMK]]

2.2.5 Buffering Constructions for X9.82

[[The issue is that if you're going to buffer entropy outside the entropy source, you have to do it in a sensible way, and we need to tell them what that is. There are some slightly tricky bits to getting this right, so we need to give them real guidance, not just "do something reasonable and you'll be okay." --JMK]]

The following constructions specify the three allowable ways to buffer entropy externally from the entropy source, for later use by the RBG mechanism. (Note that internal buffering is handled in Part 2, and can be much more flexible, as it can take advantage of knowledge about the entropy source's probability distribution.) These can be used along with the external entropy accumulation mechanisms described above, or directly on entropy source outputs, subject to the requirements in the descriptions on the differrent buffering schemes.

We specify acceptable constructions because there can be 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.

2.2.5.1 Simple Queue

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

a. It is very simple and cheap to implement.

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 incorrectly overestimating its entropy, the simple queue buffering scheme will provide no additional defense, even if the RBG mechanism is using entropy from the buffer much more slowly than it is being put in. This affects validation of the entropy source for use in the RBG, if the RBG in normal operation is expected to produce many times more entropy than is drawn from the buffer.

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

Let Q be the total number of outputs stored in the buffer. Then, entropy source outputs are fed into the "top" of the queue. If fewer than Q values are present in the queue, the queue expands to hold the new value, and all other values are "bumped down" one position in the queue. When the buffer is full, the last (oldest) entropy source output in the queue is discarded. Each queue entry carries an entropy estimate taken from the entropy estimate of the entropy source output that filled it.

When entropy outputs are requested, they are taken from the "bottom" of the queue, and the queue is updated to have one less value in it. When an entropy output is requested, and there are no outputs in the queue, either an error condition must be raised or the

request must block until there are outputs in the queue.

2.2.5.2 XOR Buffer

When we know that the outputs from our entropy source are approximate uniformly distributed, with their entropy spread among all bit positions, we can use a simple improvement to the Simple Queue scheme: The XOR buffer.

Note that this can only be used for conditioned entropy sources, and entropy sources have been accumulated by the CRC or derivation function based accumulator constructions. We assume here that entropy source outputs are all the same length.

Let Q be the total number of outputs stored in the buffer. The buffer is initialized to all zero bits for all entries. Each buffer entry has an associated entropy estimate, and all these estimates are set to zero.

When a new entropy source output is added to the buffer, the oldest entry in the buffer has the new output XORed into it; the entropy estimate for that entry is either the number of bits in the entry, or the old entropy estimate plus the new output's entropy estimate, whichever is smaller. [[Maybe we should deal with asymptotic behavior here?--JMK]]

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

When an output is requested from the buffer, the next buffer entry which has an entropy estimate of sufficient bits to satisfy the request is used. Buffer entries may be concatenated to get more entropy than exists in any single entry. As an entry is used, its entropy estimate is reset to zero, but its value remains in the buffer. If there is not sufficient entropy anywhere in the buffer to satisfy the request, then the buffer must either raise an error condition or block until enough entropy is available.

2.2.5.2.1 Using the XOR Buffer as a Conditioned Entropy Source

In order to use the XOR buffer to externally *condition* an entropy source, we simply allow the buffer entries' entropy estimates to increase up to twice the size of the entry. Only buffer entries whose estimate is twice the number of bits as are in the entry may be used as conditioned outputs.

2.2.5.3 Hash Buffer

The 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.

To add a value *inputString* with estimated entropy ee to the hash buffer, we do the following steps: (Recall that n is the block output length of the hash function.)

a. tmp = *ee*

b. While tmp>0:
(i) Let X = hash(C || buffer || *inputString*)
(ii) Let $C = C + 1$
(iii) Shift buffer right n bits, discarding the n rightmost bits.
(iv) Prepend X to the buffer.
(v) tmp = tmp - n

c. Let E=max(E+ee,R)

To pull a k-bit value from the buffer, we do the following steps:

a. If $k>E$: raise an error condition or block until entropy is available.
b. Let tmp = ""
c. While bitlength(tmp)<k :
(i) tmp = tmp || hash(C||buffer)
(ii) C = C + 1
d. $E = E - k$
e. Return least significant k bits of tmp

[[There are surely more efficient ways to do this, but there are also a few ways to walk off a cliff--my first attempt was one.]]

2.2.5.3.1 Using the Hash Buffer as a Conditioned Entropy Source

In order to use the hash buffer as an external *conditioning* routine for an entropy source, two things must be done:

a. To generate k bits of output with full entropy, we check to see that the hash buffer contains at least 2*k bits of entropy.

b. At the end, we subtract 2*k from the buffer entropy estimate.

c. If the hash buffer is used this way, it SHALL NOT also be used to provide normal entropy values.

2.3 Combining Sources and Entropy Estimates

In some RBG mechanisms, especially ones based entirely in software, many entropy sources are combined together to get enough entropy to instantiate a DRBG or to produce full entropy output.

We must start with the assumption that entropy estimates of the different sources have already accounted for any dependence between them; that is, if we have two sources that are related, the entropy estimates of the two, when added, must not "double-count" any of the entropy. Min-entropy is based on the maximum probability for a value; conditional min-entropy is based on the maximum probability for a value, *conditioned upon the other available values*. Detailed discussion of entropy estimation can be found in Part 2 of this standard. [[*PUNT* --JMK]]

Software entropy sources tend to be wildly variable in rate of entropy produced, as their unpredictability comes from the natural variability in some process that is going on only some of the time, 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.

Because of this, both accumulating and buffering entropy are extremely important in software sysems.

2.3.1 Accumulating Entropy for Output

Any of the accumulation methods described above are suitable for use with multiple entropy sources. 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 pulled from operating system statsitics and system loading sources. When performance requirements permit, the hash_df accumulation method SHOULD be used.

2.3.2 Maintaining an Entropy Pool

Any of the buffering techniques above 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 RBG mechanisms

2.3.3 Reseed Management

[[Eventually discuss Niels'/Peter's problem of multiple entropy sources and deciding which subset to trust. This is a can of worms! --JMK]]

2.4 Validation Considerations

The hardest problem in constructing a working RBG mechanism 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 very difficult. Flaws can occur all the way from basic design flaws (for example, a poor statistical model of the source's behavior, leading to consistent overestimates of entropy in every implementation of the design) to implementation problems (for example, some on-chip signal that overwhelms the effect of thermal noise in the circuit being sampled and amplified to get random bits), to physical flaws in a specific device (for example, an oscillator might stop oscillating because of a component failure) to environmental problems (for example, the circuit whose thermal noise is being amplified to provide random bits might be picking up some nearby strong radio signal).

The risk of entropy source failure is handled in three ways:

a. The entropy source design and implementation may be done in a way to minimize or eliminate 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 oscillators phase locking 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.

b. The entropy source can have 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 gets in 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 specs of the device.

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 a factor of two overestimate in the input it uses from the entropy source to instantiate for the first time, and thereafter rely on persistent state to retain enough entropy to protect against overestimates from the entropy source.

A failure in the entropy source that is not prevented by (a), detected by (b), or prevented from doing any harm by (c) is called a *critical failure* of the entropy source. A major design goal for any RBG mechanism is to keep the probability of a critical failure to a very small value. Note that we are concerned here only with failures that lead to practical weaknesses in the RBG mechanism here; an entropy source failure that leads to a DRBG mechanism which claims 128 bits of security delivering only 120 bits is not terribly important, but a failure that leads it to deliver only 80 bits of security is a critical failure. We will arbitrarily assume that a practical failure occurs when the mechanism loses 20% or more of its claimed security. Further, we will treat a basic NRBG as having a security level of 128 bits for the purposes of this validation. (Enhanced NRBGs already have a fallback security level.)

[[This 80% number is plucked out of thin air. Any suggestions for better numbers? I want to get across the idea that in validation of the entropy source and the surrounding design for using entropy, we aren't too worried about losing some small fraction of our security, but we need to keep from losing too much of it. For reference, with 20% of the strength lost, we get something like:

112 --> 90 bits
128 --> 102 bits
192 --> 154 bits
256 --> 205 bits

This all seems sensible to me. We wouldn't knowingly tolerate a *design* that claimed 112 bits of security but only gave us 90 bits, but we will accept worst-case behavior from an entropy source that does this to us, because we basically can't validate the entropy source to the same level of assurance we can validate an algorithm to. (But don't tell that to the folks who, until about a month and a half ago, were happily using MD5 in their newly-

designed protocols and crypto mechanisms.) --JMK]]

An important implication of this is that an entropy source and its surrounding operational tests can only be validated within the context of some surrounding mechanism, because different surrounding mechanisms have different ranges of critical failure. For example, a DRBG mechanism consisting of Hash_DRBG and a the ring oscillator reference source from Part 2 should be validated differently based on whether there is a seed file kept to save entropy across restarts.

[[We might also want to talk a bit about validation w.r.t. active attackers here. You need different shielding on a tamper-resistant device in the hands of the attacker than on a device intended mainly as a crypto accellerator on a well-guarded server. But I was afraid if I started writing on this right now, I'd end up talking about FIPS 140-2 module boundaries and such things. --JMK]]

3 Building a Computationally-Secure RBG Mechanism (DRBG)

An RBG mechanism that supports only computational security is called a DRBG mechanism. 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 which are computationally indistinguishable from ideal random bitstrings.

3.1 Preliminaries

A DRBG mechanism promises *k* bits of security, meaning that distinguishing the output sequence from the DRBG mechanism should be no easier than distinguishing the output sequence from an ideal block cipher running in counter mode with a *k*-bit key. Any consuming application with a *k* bit or lower security level may use a DRBG mechanism with a *k* bit security level.

The most important problem in using any DRBG algorithm is getting to a *secure state*--a state from which outputs can be generated which will satisfy the *k*-bit security level claimed by the design. In practical terms, this means that the state must be *unguessable* by the attacker. A secondary problem is ensuring that it recovers from compromise. That is, if the DRBG's state should somehow be leaked or learned through cryptanalysis, the DRBG should eventually reach a new secure state.

There are two broad categories of DRBG mechanism:

a. DRBG mechanisms with an available, live entropy source. These mechanisms rely primarily upon the quality of their entropy source to get to a secure starting state, but may also include persistent state, internal secret values, and personalization strings to provide some level of fallback security, in case their entropy source is flawed. These mechanisms are able to recover from compromise on their own, and typically do not need to have a built in limit on the total number of bits they may output or time during which they may operate.

b. DRBG mechanisms without an entropy source, which rely wholly upon secure storage

of some persistent state. These are instantiated from information provided from outside the DRBG mechanism, e.g., an output from some external entropy source or RBG mechanism. These mechanisms cannot recover from compromise without external help, and typically have a built-in cryptoperiod, limiting the number of bits generated, the length of time they may continue in operation, or both.

All DRBG mechanisms include one or more approved DRBG algorithms. The surrounding DRBG mechanism ensures that the DRBG algorithm is used correctly. Recall that all approved DRBG algorithms support three functions:

a. *instantiate*(entropy_input,personalization_string)

b. *reseed*(entropy_input,optional_input)

c. *generate*(bits, optional_input)

The DRBG algorithms will make use of whatever entropy is in the parameters (entropy_input, personalization_string, optional_input) to reach a secure state.

3.2 Construction #1: Entropy Source + DRBG Algorithm

The most natural and general way to construct a DRBG is to combine an entropy source from Part 2 with a DRBG algorithm from Part 3. While the basic notion of how to do this is described in Part 3, and is relatively straightforward, there are some important considerations to consider with this kind of design.

There are three times when the DRBG mechanism must request entropy from its entropy source: When instantiating the DRBG, when reseeding it, and when generating an output sequence with *prediction resistance*. The goal for each of these three uses of entropy is to get the DRBG algorithm to a secure state, even if it 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.

A major goal of the design of the DRBG mechanism is to ensure that the DRBG algorithm benefits from the entropy available to it as much as possible.

3.2.1 Instantiating the DRBG Algorithm

Before the first bit of output is generated from the DRBG mechanism, the DRBG algorithm must be instantiated.

Instantiating the DRBG algorithm requires two parameters: the entropy input and the personalization string.

3.2.1.1 The Oversampling Factor and the Entropy Input

Let w be the *oversampling factor*. The oversampling factor is used to decide how much entropy to request from the entropy source for instantiation. Any DRBG with a k-bit

security level must be instantiated with at least k+64 bits of min-entropy, but a mechanism with an oversampling factor of w will instantiate with w*(k+64) bits of min-entropy from the entropy source.

The value of w affects validation of the entropy source used; a higher value of w means that the entropy source must fail in a more drastic way to cause a *critical failure*.

3.2.1.2 The Personalization String

The *instantiate* function also allows a personalization_string--an input which can contain additional entropy, but is mainly intended to differentiate this DRBG instantiation from all the others that might ever appear. The personalization_string SHOULD be set to some bitstring which is as unique as possible to a specific implementation or instance of a DRBG mechanism, and MAY include secret information. The personalization_string can include any secret information whose value is no greater than the claimed strength of the DRBG, as the DRBG's cryptographic mechanisms (specifically, its backtracking resistance and the entropy provided by the entropy source) will protect this information from disclosure. Good choices for the personalization_string contents include:

a. Device serial numbers
b. Public keys
c. Userids
d. Private keys
e. PINs and passwords
f. Secret per-module or per-device values
g. Timestamps
h. Network addresses
i. Special secret key values for this specific DRBG instance
j. Seed file contents (see next subsection)

The personalization string has several effects on the security of the DRBG mechanism:

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 mechanism'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, 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 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 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 *k* 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 mechanism lifetime)/2.

3.2.1.3 Saving Entropy Across Startups: The Seed File

Some DRBG mechanisms (especially all-software ones) may have entropy sources that are not reliably available in time to instantiate the DRBG algorithm before it is needed. Others may have an entropy source which 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 mechanism can save some entropy across startups, using what we will call a *seed file*. (Note that the use of the word *file* here is not meant to imply any particular way of storing the entropy.) Any DRBG mechanism that can afford the added cost of secure, nonvolatile storage for a seed file SHOULD use one.

When we have a seed file, the first instantiation of the DRBG algorithm SHOULD have more demanding parameters than other instantiations. This means:

a. The entropy source SHOULD be extensively tested before the first instantiation, but MAY require much less testing for later instantiations.

b. The oversampling factor *w* SHOULD be set to a much higher value, e.g., 8 or 16, but MAY be set to 1 for all future instantiations.

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 availalble, plus the current contents of the seed file.

c. The DRBG algorithm is instantiated.

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

3.2.2 Reseeding and Prediction Resistance

Reseeding a DRBG algorithm means adding entropy input (and an optional input) into the DRBG 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.

3.2.2.1 When Does a Reseed Occur?

The DRBG algorithm may be reseeded for three reasons:

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

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

c. The DRBG mechanism carries out a reseed based on available entropy and its *reseeding strategy* to minimize the exposure due to any compromise of the DRBG state.

The DRBG mechanism requests at least k bits of min-entropy from its entropy source, and provides this as entropy input to reseed the DRBG algorithm. Additional information which 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 optional input to the reseed function.

3.2.2.2 Seed Files and Reseeding

If a seed file is present, it SHOULD be used in reseeding. If it is used, it SHALL be used in the following way:

a. The seed file contents are included as part of the optional input to the reseed function

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

[[This raises some problems for composite mechanisms, which I'll hopefully address in the enhanced NRBG section, below. --JMK]]

3.2.2.3 Reseed Management: Using Entropy Wisely in a DRBG Mechanism

Most DRBG mechanisms with an entropy source available will 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 mechanism can attempt to protect the underlying DRBG algorithm from as-yet-unknown cryptanalytic attacks, as well as from any other form of compromise.

The DRBG mechanism collects and buffers entropy from the entropy source over time. At some point, the mechanism 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 DRBG mechanism 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 DRBG mechanism can give the DRBG algorithm the best possible chance of recovering from compromises of its state, and of resisting unknown cryptanalytic attacks.

3.2.2.3.1 Reseed Management Before the First Output is Generated

Any DRBG mechanism with a live entropy source SHOULD implement the following simple piece of reseed management:

a. The mechanism 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 to a secure state before the first output is generated.

3.2.2.3.2 Reseed Management After the First Output

After the first output, the proper form of reseed management 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 more and more entropy 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.

A simple reseed strategy that provides some support for both of these is as follows:

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

a. Each time the fast pool's assessed entropy is k bits, the DRBG mechanism uses its contents to reseed the DRBG algorithm. If a reseed is triggered by the DRBG algorithm or the consuming application, the fast pool's contents are always included, along with whatever entropy source outputs are required to reach an assessed value of at least k bits of min-entropy for the reseed.

b. The slow pool's reseed threshhold starts at $t = 2*k$. Each time the slow pool reaches t bits of assessed entropy, the DRBG mechanism does the following steps:
(i) Reseeds the DRBG algorithm with the slow pool's contents.
(ii) Sets $t = 2*t$.

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

3.2.3 Generation of Outputs

The DRBG mechanism simply calls the DRBG algorithm to generate outputs.

3.3 Construction #2: External Seed Source + Persistent State

A DRBG mechanism can also be built without access to a live entropy source. This DRBG mechanism cannot support reseeding or generating outputs with prediction resistance, and SHALL have a fixed cryptoperiod, which may be stated in terms of maximum number of outputs allowed, or of maximum time until the mechanism must be discarded or re-instantiated.

This construction requires a DRBG algorithm and persistent, secure storage. [[Should we discuss how to do that using crypto, and the limitations of that?]] Instantiation of the DRBG algorithm requires interaction with an external entropy source or RBG.

3.3.1 Instantiation

The DRBG algorithm requires an entropy input to instantiate. The entropy input is provided from outside the DRBG mechanism, over some secure channel, so that it cannot be intercepted or altered by any attacker. The entropy input may come from another RBG mechanism, an external entropy source, or conceivably even a human-driven entropy source such as flipped coins or rolled dice.

3.4 Construction #3: Chaining RBG Mechanisms to Provide Seed Material

[[We need to talk about this, and here's the place to do that.--JMK]]

...

3.6 Conclusions

///// Old outline begins

2 Providing Computational Security with DRBG Mechanisms

2.2 Instantiation [Known]

Explain the goal of instantiation, and what can go wrong. Explain that using K+64 bits of min-entropy in instantiation is the minimum that you can do, but that it's much better to use more. Get across the idea that instantiation is the time to get it right--there's relatively little value in worrying about recovery from compromise, until you've done everything possible to start out in a secure state.

2.2.1 Personalization Strings [Known]

Explain personalization strings. Show how to construct them, and explain the situations in which they become important for security. Use the analogy of salt in password hashing schemes.

Explain the special requirements for personalization strings when used with a conditioned entropy source and no derivation function.

2.2.2 Seed Files [Known]

Explain seed files, how they can be generated off-chip and used to accumulate entropy over multiple startups. Discuss how they should be protected and guidance on how they are to be managed.

Discuss the difference between keeping a seed file, and keeping the whole DRBG in persistent memory across startups.

Explain the specific requirements for seedfiles when used with a DRBG without support for derivation functions

2.2.3 Startup Testing and Security Strategy [Known]

The goal is to get as many bits from the entropy source into the DRBG as possible. This can be done through an instantiate and many reseeds, or through one big instantiate. Explicitly allow the us of entropy bits for statistical testing at startup for an instantiation or a reseed.

2.3 Reseeding and Prediction Resistance [Known]

Explain the two purposes of reseeding:

a. To recover from a weak state or a compromise.

b. To limit the effective cryptoperiod of some DRBG seedlife/instance/whatever.

Explain the notion of catastrophic reseeding, and why it's better than dribbling entropy in. Explain the basic requirement of reseeding entropy.

Explain the special requirements on the reseeding operation when done with a conditioned entropy source. (For the block cipher DRBGs, there are no special requirements; for other DRBGs, there's no method yet defined for doing this, but we should talk it over....)

2.3.1 Reseeding and Seed Files [Known]

Explain that it makes sense to generate a fresh output to use to replace the seed file content immediately after reseeding, if possible.

2.3.2 Additional Input in Reseeding [Known]

Explain the purpose and value of additional input in reseeding, as a way to provide entropy from the application, or inputs which may

be unknown to at least some attackers, even if not all attackers.

Explain how this is done when using only conditioned entropy sources. (No way at present, but it's easy to add this!)

2.3.3 Reseeding Strategies [Known (mostly; there are a few dark corners here.)]

Discuss:

a. Maintaining a pool which holds accumulated entropy, and which is used to reseed whenever the pool is "full."

b. Maintaining a pool which accumulates entropy from instantiation until the first output is requested, and reseeds before that first output to maximize the chances of getting to a secure state.

c. Maintaining a pool which accumulates entropy continuously, and which reseeds before each new output if its entropy estimate is sufficient.

d. Split pool designs like Yarrow, and theoretically interesting designs like Fortuna, for providing a kind of worst-case guarantee.

///// Old outline ends

4 Information-Theoretically Secure RBG Mechanisms: NRBG Constructions

An information-theoretically secure RBG mechanism produces bits that are indistinguishable from random, even given unlimited computing power. In other words, the bits output from this kind of mechanism 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. In X9.82, we often refer to a mechanism for producing ideal random bits (bits with information-theoretic security) as an NRBG.

4.1 Preliminaries

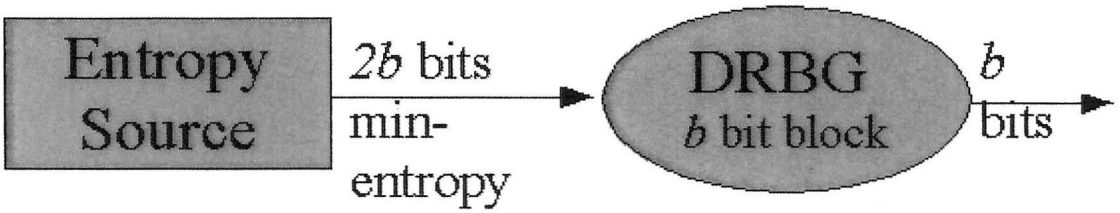
Information theoretic security can be provided only by the entropy source. Since entropy sources are generally a lot less reliable than deterministic components, a fundamental question to ask about this kind of construction is "what happens when the entropy source fails in some undetected way?" There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

a. *Enhanced NRBGs* guarantee a fallback to an approved DRBG, should the entropy source suffer some kind of disasterous failure. This has the practical effect of making validation of the entropy source much easier.

b. *Basic NRBGs* make no guarantee of a fallback to an approved DRBG. This means that validation of the entropy source must be much more demanding.

4.2 Enhanced NRBGs

4.2.1 Construction #1: Using a DRBG as an External Conditioning Function



In this section, we describe how to use any approved entropy source to produce an NRBG whose security rests primarily upon the entropy source. Each DRBG has a natural *block size*--the size of its internal, cryptographic component's inputs or outputs, whichever is smaller. For hash function based DRBG algorithms, the block size is the hash function output size; for block cipher based DRBG algorithms, it is the block size of the cipher, and for number-theoretic algorithms, it is the size of the internal state--the point on the curve or the large integer that makes up the secret part of the working state. We will call the block size *b*.

4.2.1.1 State of the Enhanced NRBG

The enhanced NRBG consists of two components:

a. A DRBG algorithm and associated state from Part 3, with blocksize *b* as discussed above.

b. An entropy source from Part 2.

4.2.1.1 Initialization

As with all our constructions, we strongly recommend instantiating the DRBG with more entropy than the minimum, as this allows for easier validation of the underlying entropy source for use in this system. All of the techniques for improving the security of instantiation in computationally-secure RBG mechanisms can be reused in NRBG mechanism, and will affect the validation process for the RBG mechanism in question. We especially recommend the use of persistent state for the RBG, so that entropy source failures in the field will not lead to a catastrophic loss of security.

The DRBG algorithm component associated with this enhanced NRBG construction MAY have an additional long-term secret *S*. If it is present, it SHALL be used as an input with which to reseed the DRBG algorithm after the normal instantiation process has happened, but immediately before generating any outputs from it.

Let *w* be the oversampling factor for instantiating the DRBG algorithm component, where *w* must be at least 2. We now do the following:[[Should we let $w=1$?]]

a. Request $w*(k+64)$ bits of min-entropy from the entropy source, and put it into *seed*..

b. Instantiate the DRBG algorithm with *seed* and any personalization string information

that is available. At this point, seed files and other material can be used, as described above.

c. The Enhanced NRBG construction is now ready to generate outputs.

4.2.1.1.1 Using a Conditioned Entropy Source

When the entropy source is a conditioned entropy source, and the DRBG does not include a derivation function, a single input seed can only contain as many bits of min-entropy as the internal state size. In that case, we do the following steps:

a. n=0

b. while $n<w*(k+64)$:
(i) Reseed DRBG algorithm using conditioned entropy source, furnishing *seedlen* bits.
(ii) $n = n + seedlen$

c. The Enhanced NRBG construction is now ready to generate outputs.

4.2.1.2 Generation

When outputs are requested, our full-entropy requirement says that we must request enough entropy to make the output full entropy, or at least indistinguishable from full entropy even with infinite computing power. In order to accomplish this, we do the following:

a. Start with an empty buffer, *tmp*, and a number of requested bits n.

b. Until we have generated enough bits to satisfy the request, repeat the following steps:
(i) Let seed = at least $2*b$ bits of min-entropy from the entropy source.
(ii) Let x = b bits of output generated from the DRBG, using *seed* as additional input
(iii)Let $tmp = tmp \| x$

c. Return *tmp* truncated to the requested number of bits.

[[This never triggers the reseed mechanism; should we force the first one to be a reseed? We shouldn't have to do a reseed between every output (this is rather expensive!). Comments? --JMK]]

4.2.1.3 Pseudocode

In this section, we give a quick description of the NRBG construction in Python-influenced pseudocode.

```
class Enhanced_NRBG_1(object):
    def __init__(self,entropy_source,drbg_algorithm,w,b,personalization_string=""):
            self.drbg = drbg_algorithm
            self.source = entropy_source
```

```
            seed = self.source.get_entropy(w*(self.drbg.security_level+64),1,99999)
            self.drbg.instantiate(seed+personalization_string)
            self.ready = 1
    def generate(self,bits,additional_input):
            tmp = ""
            while bitlen(tmp)<bits:
                seed = self.source.get_entropy(b,1,99999)
                tmp = tmp || self.drbg.generate(b,seed)
            return truncate(tmp,bits)
```

4.2.1.4 Validation Concerns for the Entropy Source

This design ensures information-theoretic security if the DRBG algorithm is a good conditioning function for the entropy source, and if the entropy source is behaving as expected. It guarantees computational security if the DRBG is instantiated to a secure state before any outputs are generated.

The entropy source validation must ensure two things:

a. In the normal, expected operations of the entropy source, the requests for entropy must be fulfilled as requested.

b. The probability of a *critical failure* must be acceptably low. In this system, validation will be primarily concerned with ensuring computational security that is within 20% of the claimed number of bits of the DRBG security level. The first output has been fed at least $w*(k+64)+2*b$ bits of min-entropy, according to the entropy source. The probability of a *critical failure* of the entropy source is the probability that, in the first $w*(k+64)+2*b$ bits of output, there will be fewer than $0.8*k$ bits of entropy, without causing the startup or continuous tests to fail.

4.2.1.5 Making a Composite Mechanism

The 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 critical operations like keypair generation, and relatively low-importance operations like generating nonces, TCP sequence numbers, IVs, etc. If this mechanism allows access to its underlying DRBG, it SHALL be done as follows:

A request for computationally-secure bits is responded to 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 *seed*.

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

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.

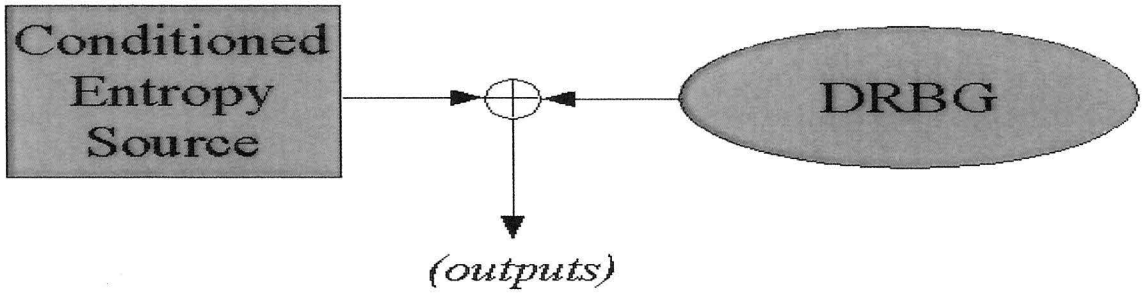
As an example, suppose you first generate a full-entropy output (pump 256 bits of min-entropy into AES-128-CTR, and then produce 128 bits of full-entropy output). I then request a few hundred bits of output from the *computationally secure* side. If you didn't reseed first, and I was able to do a little 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.

--JMK]]

4.2.2 Construction #2: XOR NRBG Construction

Given a conditioned entropy source and a DRBG, this NRBG construction works by first instantiating the DRBG algorithm, and then 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 XORing them together.



This construction is quite simple, and it is easy to see why the output sequence can be no less secure than the stronger of the two input sequences, so long as the two sequences are independent. However, this does require access to a conditioned entropy source. If such a source is not available

4.2.2.1 Instantiation

4.2.2.2 Generation

4.2.2.3 Periodic Reseeding

4.2.2.4 Providing Composite Output Access

4.2.2.5 Validation Concerns

This is, to my mind, the simplest design. If the probability model is right, the NRBG is information theoretically secure; if not, it's no worse than the DRBG.

We need to explain:

a. The time before outputs are generated, where we keep spinning the design as fast as the entropy source will support, and reseeding from the outputs. We want to maximize our chances of getting to a secure state, so all our discussion of things like seed files and personalization string applies here.

b. The time when outputs are requested, where we crank out outputs as requested, limited by the conditioned entropy source only.

c. The time between outputs, when we occasionally "turn the crank" and collect more entropy by generating more outputs and reseeding with them.

d. A discussion of fixed secret keys and seed files that can be included in the DRBG Instantiation to provide the required security properties.

e. A discussion of how the design may be used as a DRBG by simply generating outputs from the DRBG and not XORing them with the conditioned entropy source's outputs.

f. Some notion of how we can power-down the system safely, perhaps including a small amount of buffering (like buffering the next output to be generated, to give us time to start back up our entropy source and get out of startup transient behaviors before we start using it again).

4.3 Basic NRBGs

In X9.82, we define a *Basic NRBG* as an RBG claiming information-theoretic security, which does *not* guarantee at least the same security as some approved DRBG algorithm. This kind of construction falls into three broad categories, with quite different security properties and similarly different reasons for a designer to choose them. The simplest

constructions provide raw access to some conditioned entropy source's output bits. This has the advantages of extreme simplicity, and potentially very low gate count. (For example, an RFID which needed random numbers might sensibly use such a construction.) However, this kind of design has very little protection from failures in the entropy source; even minor flaws that would have had little effect on the cause

4.3.1 Raw Access to Conditioned Bits

4.3.2 Conditioning With Cryptographic Mechanisms

4.3.3 Conditioning or Combining With an Unapproved DRBG Algorithm

5 Composite Designs [[Will I have covered this?]]

Explain what this is all about--basically, how we can combine multiple raw or conditioned entropy sources together safely, to get a result that's no worse than the better of them.

This can be used either:

a. To allow use of unapproved entropy sources along with approved ones. (If you're convinced you have a better one than the approved stuff, we want you to use it.)

b. To get added assurance, when one entropy source is deemed unreliable. This might be a good strategy for the Basic NRBG.

Raw Entropy Sources and Concatenation [Known]

If we want to combine raw entropy sources in a single Instantiate or Reseed call, we have to concatenate them.

Conditioned Entropy Sources and XORing [Known]

For conditioned entropy sources, we can XOR or concatenate, whichever is more reasonable. But the sources MUST be independent.

Serial Reseeds and Additional Input [Known]

We can always instantiate with one entropy source, and then reseed with another. And we can always stick unapproved entropy source output into a DRBG as additional input.

6 Combining RBGs

Combining RBGs might be done for a number of reasons, including:

a. The desire to use an unapproved DRBG believed to be superior in security with an approved DRBG, getting the benefit of each.

b. The desire to combine DRBGs or NRBGs driven by different

entropy sources for increased assurance.

c. The desire to combine RBGs implemented by different people or contained on different modules, for increased assurance.

There is only one acceptable way to do this: By XORing the RBG outputs together. This imposes the additional requirement that the RBGs be seeded independently.