Comments on *X9.82 Discussion Issues for August 11, 2004*
My comments are in regular type. --JMK

General Issues:
4. How does prediction resistance relate to the case where entropy is dribbled in?

There's no open issue here, it's just counting. Suppose I start with knowledge of your state, and then you update the DRBG with a sequence of entropy inputs, and finally generate another output. Can I distinguish that next output from random? (On a more practical note, given the first 128 bits of output, can I guess what the other bits will be?)

The answer depends only on whether or not I can guess that sequence of entropy inputs. When the sequence of entropy inputs has provided at least K bits of min-entropy, where K is also the security level of the DRBG, we know that I can't guess the new state given knowledge of the old state—because that's no easier than guessing a K-bit key, and I can't do that.

If we dribble entropy in and want to support prediction resistance, we can count the amount of estimated entropy dribbled in since the last output. If that's K bits or more, then we can generate with prediction resistance; otherwise, it's not.

I know this K is different than the $K+64$ we're going to be requiring for instantiation; and there's a reason for this (we can consider the starting DRBG state before the added entropy as a kind of personalization string that's unique per DRBG instance).

6. Restructuring: Part 2 to discuss basic NRBG/conditioned entropy source/entropy source? Construction of an NRBG other than in Part 2? Drop the distinction between DRBGs and NRBGs?

I think the basic idea proposed was that there should be a clear break between entropy sources, DRBG algorithms, and construction methods for using these components to get the desired set of properties. That probably includes the two enhanced NRBG constructions, as well as reseeding guidance for DRBGs, and guidance on how to combine trusted and untrusted components together in a system.

9. Remove the 80 (and possibly the 112) bit security level?

I think it would be nice to get rid of the 80-bit security level, but that this is probably a little unreasonable, since we're allowing things to use it till 2015 or so. I'd like to at least strongly urge designers to move to at least the 112-bit security level.

Part 1 Issues:
3. Can min_entropy be determined if there is no independence? More guidance in the text for when correlation exists.

Conceptually, yes—we just look at the largest *conditional* probability instead of just the largest probability. But in practice, this is a lot harder, because it's hard to know how to compute the best conditional probability—you have to have some kind of model that

spells out the dependence. But if our model is wrong, we can be way off. This seems no different for other measures of entropy. For example, both guessing entropy and Shannon entropy also need to be computed using conditional probabilities if the output probabilities aren't independent, and they raise the same set of problems. It seems to me that it's always easier to collect data to find the one highest probability than to find all or most of them, but I don't have a proof for that.

Part 1 agreed upon changes:

1. Enhanced NRBGs are "computationally bounded", rather than having an infinite security level. Basic NRBGs are an informationally secure source of random bits. DRBGs are computationally secure.

A DRBG provides security against computationally bounded attackers, and the DRBG's security level determines what the required computational bound is.

An NRBG provides security against computationally unbounded attackers, assuming:

a. The entropy source is behaving properly
b. Whatever additional algorithm assumptions come from the conditioning function are true.

Providing security against unbounded attackers means there's no security level associated with the NRBG; instead, any NRBG can support any security level.

An *enhanced* NRBG is an NRBG that also provides security against computationally bounded attackers, even if the entropy source fails, so long as the internal DRBG has reached a secure starting point.

An additional distinction exists between the two enhanced NRBG constructions. The continuous-reseeding construction is probably easier to implement and more flexible, but imposes some very mild assumptions about the underlying DRBG mechanism, which could undermine the unbounded security level if they proved false. The XOR construction is a little harder to implement and a little less flexible, but imposes no assumptions about the underlying DRBG mechanism when the entropy source is operating properly. This distinction is very small, and may not be relevant to anyone.

The continuous reseeding construction's security is based on the concept of *unicity distance*. That is, the DRBG produces no more output than would be required to uniquely determine the input entropy string. (It may have to produce a little less than this, to get all the required properties for an NRBG.)

7. Explicitly say that, for DRBGs and enhanced NRBGs, entropy is precious (recycle entropy).

I think this is handled pretty well within the DRBG algorithms—adding entropy in any way doesn't cause us to throw anything away. We can still give guidance for building entropy pools for NRBG designs and for more general entropy sources, but some implementations may want to limit the amount of RAM they have to use for buffered entropy source output, or may not want to have to implement any kind of a mixing

function.

8. Characterize mixing functions for preserving or accumulating entropy.

Right. I was discussing this in my entropy sources presentation, and in an e-mail I sent to Mike before that.

10. Add a DRBG with a continuous reseed capability (see slide 69).

I think this is just telling people how to use our existing DRBG algorithms in this way. It's much more a mode of use than a new algorithm.

Part 2 Issues:
1. Include a discussion of NRBGs that are in the literature?

This is a really good idea.

For anyone who's interested in seeing a reasonable first cut at what validating an entropy source should look like, check out the full report on the VIA C3 RNG. (Google for it, but make sure you get the 40-page-or-so full report, not the 5-page-or-so summary.) I wish they'd make some of their modeling and such a little more explicit, but it's a reasonable attempt, and I'd feel quite safe using the RNG as an entropy source in normal ways, with their worst-case assessment of .75 bits of entropy per bit of output.

There's also the Intel RNG report, and a few papers out there on designing or analyzing cryptographic entropy sources or RNGs. The practical ones always seem to be based on either free-running oscillators or some kind of noisy analog component and an amplifier to provide enough gain to get thermal and shot noise, and they always seem to be able to support really huge numbers of bits per second of both output and entropy.

I need to send Mike some bibliography entries for Part 2. That might really be one of the more important parts of the document!

4. Sample at a higher rate to test the entropy sources? Fast sampling may be preferable for health testing. Is anything different if the entropy rate is high?

This is partly from an Australian statistician's paper analyzing a bunch of hardware RNGs. In order to guarantee to himself that he wasn't getting some kind of PRNG outputs, he changed the sampling speed (I'm not sure whether he controlled the clock signal to the devices), and verified that sampling faster made the statistics get worse.

7. Use lots of independent entropy sources? Evaluate at least one? Test entropy sources, if possible.

This is an interesting can of worms. Niels explained the problem he's running into at Microsoft: He basically can't assume *anything* about a configuration, because their OS goes so many places. He is often using information from drivers about which he knows very little ("here's the Ubercorp USB optical mouse driver, good luck!"), and he still has

to see if he can get his DRBG to a secure state.

The basic problem from a validation point of view is that if you aren't able to make *any* assumptions about your sources, it's hard to work out how you can ever comply with any meaningful requirements. When you start generating bits, you're implicitly assuming that you have a secure starting point for your DRBG. But maybe you've just seeded from an entirely deterministic set of events, because you don't know that the storage devices are all flash disks, the keyboard and mouse events are coming over an internet connection, etc.

8. Allow the testing of the output of a conditioning function? May provide a "sense" of the entropy source randomness or bias.

This is already discussed, I think. For conditioning functions that are very closely bound to the probability model of the underlying nondeterministic mechanism, this is potentially useful as a way to catch some kinds of problems with the entropy source itself. (The idea is that there are a huge set of off-the-shelf statistical tests for independent uniform unbiased bits.) On the other hand, for hashing-type conditioning functions (stuff like SHA1 or a 32-bit CRC), which aren't really dependent on the probability model of the nondeterministic mechanism, I don't see much point in testing the output. (The extreme case is running statistical tests on SHA1 outputs, which is never going to detect anything so long as the input to SHA1 is at least a little different for each output.

13. Set appropriate P-values for health testing? Decision based on the number of entropy sources?
14. Allow/require multiple layers of testing?
15. Specify the probability of a failure?

Just to complicate things a bit, how do we deal with the kind of tests where we're looking for a known or suspected failure mode?

It seems like there's a straightforward approach here:

a. Decide on an acceptable probability of a false positive in the tests.

b. If a single test ever has a P-value lower than that acceptable probability, fail immediately.

c. Combine p-values of all the tests; if the P-value that results is less than that acceptable probability, fail immediately.

d. Combine that combined p-value with as many others as we still have stored. Again, if the combined P-value is lower than that acceptable probability of false positives, then fail.

e. For marginal scores (e.g., P-value of 0.001 or something), we can also retest. However, I think we need to work out whether this will cause some problems with the P-

value combining algorithms. (That is, if the number of tests I run depends on some of my P-values, is this going to invalidate my combined P-value number?) This seems like a great question for the statisticians.

20. Low false positive rate. If P = 10-4, retry 3 times? Use adaptive tests (layers of tests: health tests, followed by sick tests if there are failures)? Finish as soon as you pass. establish a final error point (absolute limit).

I think there were two ideas here:

a. Repeat tests with low P-values, and note repeated falures somehow.

b. If we get a suspicious value on one test, we may want to decide (internally and automatically) to run other tests. For example, suppose we have a ring oscillator based source. We might first sample a few thousand bits, and run monobit, runs, and poker tests on the outputs. (The expected scores have to be modified for the distribution we actually see from the ring oscillator bits, which probably won't be perfect random-looking bits, but this is really easy to do.) If we get scores in line with what we expect (a P-value less than, say, 0.01 on the combined scores, where

P-value(score) = probability(score | original ring oscillator distribution)

then we trigger a much more extensive set of tests. (I chose those three tests because they're fast, and I can see ways to adapt them to a known distribution without too much trouble.)

Along with this, we need to keep in mind that some P-values are so low, we must immediately signal a failure. Like, if we get a long stream of zeros out of our ring oscillator source, it probably means the thing isn't even oscillating anymore.

22. For enhanced NRBGs: if the entropy source fails, but the DRBG is still OK, the decision to continue depends on application requirements?

I'm a little worried about this. Some tests may be accumulating evidence of failure over a long time, so that the final failure condition is due to combining the last ten sets of combined P-values from startup testing. In that case, how can we trust that our entropy source has been working as we expected? On the other hand, I see the problem for systems that have a working seed file they've been keeping from their initial startup, years ago, and which have just now noticed that their oscillators aren't oscillating anymore.

23. Condition to fewer bits?

What does this mean? Conditioning based on "hashing" (in the computer science sense, not necessarily the cryptography sense) needs some extra bits of entropy. Basically, each input string is assigned a random output value. If we had exactly 1024 input strings of equal probability, and exactly 1024 possible outputs, this random assignment of outputs would mean that about 1/e of those outputs would not be possible. We have to have enough extra input values that we expect those outputs to be very close to a uniform

distribution.

Now, this seems to get kind-of ugly, and I think we end up with this bound that says we can't do conditioning by hashing without requiring twice the min-entropy in the input as we want uniform independent unbiased bits in the output. This basically happens because the min-entropy specifies only one probability in the distribution, and if the others are all really small, we can get this thing happening where the one big probability gets assigned to one output value by the hash, and the sum of all those other probabilities doesn't have a chance to even out the output distribution.

I need to think about this a bit more, to see if we can do better given other assumptions. (Like, when we know the whole probability distribution, can we make our conditioning-by-hashing more efficient? Of course, given the whole distribution, we can make our conditioning perfectly efficient....)

1. Parameterize the security assumption, rather than using 2^{64}.

I'm not sure how much this should be parameterized. The three sensible suggestions were:

a. Maximum innocent operations = a free parameter, the implementor can choose his own within broad guidelines (say, >=2^{32}, <= 2^{security level}).

b. Maximum innocent operations = some function of security level, e.g., 2^{security level/2}.

c. Maximum innocent operations = a parameter that's specified per DRBG, and so different DRBGs can specify different limits. It might be sensible to do this so that TDEA-based designs can keep mio=2^{32} or something, while 256-bit security level designs get mio=2^{80} or something.

I really dislike adding more parameters, but if we have to do this, then item c) is my first choice. Were you guys convinced by the arguments toward making this a parameter that's changeable by the designer or implementor, or chosen by the user at instantiation time?

7. Use different counters for the block cipher derivation function?

I don't understand this item.

10. Dual_EC_DRBG: Truncate more bits and reseed less frequently? Are there correlations among bits? Allow a conditioned entropy source instead of a hash derivation function?

It wouldn't be crazy for all our DRBGs to support either derivation function outputs or conditioned entropy outputs equally in their instantiation and reseed operations.

14. DRBG boundary vs. cryptomodule boundary:

I think a more complete definition of this term in the glossary would make things a little easier to understand. Even after reading the current glossary entry for this term and all of our discussions, I am still not sure I understand exactly what you mean by "DRBG boundary." If we all agreed on a definition, then we could make some kind of headway on working out what the requirements are.

Part 3 agreed upon changes:
3. From John's slide 12: allow 264 bytes between reseeds. Eliminate counters, where possible.

Not exactly. We want to limit both Generate calls per reseed, and also bytes produced per Generate call. Suppose our maximum innocent operations ("miops?") is M, the maximum number of output bytes per Generate call is G, and the maximum number of Generate calls per Reseed is Q. Then, we know two things starting out:

G * Q <= M (approximately—each operation produces more than one byte, except possibly for the number theoretic DRBGs)

The DRBG itself can give us limits on G and Q. For example, the tightest bounds come from TDEA-OFB, where we've set both of those to 2^{16} for reasons related to TDEA's 64-bit block size.

The problem with these parameters is that they kind-of make the DRBG unusable. An application should be using one Generate call each time it needs to do something involving the DRBG. It's crazy to tell someone they can only use their DRBG 2^{16} times in their device's lifetime, but if they don't have an entropy source onboard, that's what we're telling them with these parameters.

There are three problems, all basically related to block size issues:

a. If we let one Generate request run too long, its output becomes distinguishable from random, because it never allows an 8-byte block of output to repeat. If you're given either a 2^{32} block output sequence from TDEA-OFB, or an ideal random string, you can decide which you've been given with a pretty good probability, because you expect at least one repeat with probability a little better than half for the random string, and if that occurs, you know it didn't come from TDEA-OFB. This is a purely academic issue.

b. If we let one Generate request run too long, the probability of a short cycle grows. It never gets big, but with 2^{32} outputs, the probability of a short cycle is about 2^{-32}. (The probability of a short cycle on the N+1th output, assuming one hasn't occurred yet, is 1/(2^{64}-N-1).) This isn't likely to happen, but if it does, it's a big practical problem because outputs start repeating!

c. Both of these effects get multiplied by the number of Generate requests. So, if we had 2^{32} Generate requests of 2^{32} 8-byte blocks each, we would both have a trivial distinguisher from random for this DRBG, and a serious chance that we'd see an internal collision in OFB-mode that would lead to a bunch of easily-predicted output bits.

Now, when we restrict outputs to 2^{16} bytes = 2^{13} blocks, we keep those probabilities much lower: Down around 2^{-39} probability of a pair of output blocks repeating in each Generate output stream. When we only see 2^{16} such outputs, the distinguisher is very weak. (Given a whole output sequence from the DRBG or a random string of the same size, we have about a 2^{-26} probability of being able to immediately decide that it's a random string instead of the DRBG outputs.)

Similarly, when we restrict outputs to 2^{16} bytes = 2^{13} blocks, we have about a 2^{-51} probability of hitting a short cycle in OFB-mode, and so over 2^{16} Generate requests, we have a still negligible 2^{-35} chance of hitting a short cycle in any one of these requests.

So long as we keep these two probabilities low enough, we can move around the specific parameters. I think the only sensible thing to do is to make the probability of a disasterous internal collision be the thing we worry about most, and I'd like that probability to be very low. For TDEA-OFB, this is the kind of parameters we can get (with acceptably low distinguishing probabilities, as well).

Output bytes Per Generate	Lg(Calls Per Reseed)	Lg(prob of Distinguisher)	Lg(prob of Disaster)
16	48	-15	-15
16	40	-23	-23
16	32	-31	-31
64	40	-19	-21
64	32	-27	-29
256	40	-15	-19
256	32	-23	-27
1024	40	-11	-17
1024	32	-19	-25
4096	32	-15	-23
4096	24	-23	-31
65536	16	-23	-35
65536	24	-15	-27
65536	32	-7	-19

I think the sensible points here are either 2^{16} output bytes, and 2^{24} Generate requests, or 1024 output bytes, and 2^{32} Generate requests. Comments?

10. Request entropy up to the "state" size.
11. min_entropy = requested_strength + 64.

This all needs to be hashed out; I'll have some text by the meeting.

--John