From:	Kelsey, John M. (Fed)
To:	Bassham, Lawrence E. (Fed)
Cc:	Moody, Dustin (Fed)
Subject:	Re: Here"s text summarizing what we said in our meeting. Note that John will need to expand on his advice regarding "seed expander"
Date:	Wednesday, July 19, 2017 5:16:02 PM

Is there anything about this guidance we no longer agree with? It seems like the actual guidance we want to give now is more-or-less the same. Specifically, we want randombytes() to be AES CTR DRBG, we provide an AES seed expander that we explicitly say isn't random-oracle-like but should work when the key is unknown, and we say to use KMAC when random-oracle-like behavior is needed.

Is there anything else we need to specify?

--John

From: "Bassham, Lawrence E (Fed)" <lawrence.bassham@nist.gov>
Date: Wednesday, July 19, 2017 at 3:11 PM
To: "Kelsey, John M. (Fed)" <john.kelsey@nist.gov>
Cc: "Moody, Dustin (Fed)" <dustin.moody@nist.gov>
Subject: Re: Here's text summarizing what we said in our meeting. Note that John will need to expand on his advice regarding "seed expander"

John,

I'm just checking… Are you asking this to make sure you cover everything and we are all on the same page before doing the write-up we are looking for?

Larry

From: "Kelsey, John M. (Fed)" <john.kelsey@nist.gov>
Date: Wednesday, July 19, 2017 at 1:52 PM
To: "Moody, Dustin (Fed)" <dustin.moody@nist.gov>, "Alperin-Sheriff, Jacob (Fed)" <jacob.alperin-sheriff@nist.gov>, "Bassham, Lawrence E (Fed)" <lawrence.bassham@nist.gov>, "Chen, Lily (Fed)" <lily.chen@nist.gov>, "Jordan, Stephen P (Fed)" <stephen.jordan@nist.gov>, "Liu, Yi-Kai (Fed)" <yi-kai.liu@nist.gov>, "Miller, Carl A. (Fed)" <carl.miller@nist.gov>, "Peralta, Rene (Fed)" <rene.peralta@nist.gov>, "Perlner, Ray (Fed)" <ray.perlner@nist.gov>, "Smith-Tone, Daniel (Fed)" <daniel.smith@nist.gov>
Subject: Re: Here's text summarizing what we said in our meeting. Note that John will need to expand on his advice regarding "seed expander"

Everyone,

This is the guidance I sent out on the PQC forum before for seed expanders. The big picture was:

1 The calls to randombytes() should really be generating outputs from AES CTR DRBG.

2 If you need cryptographically strong pseudorandom bits and you start with a secret plus some other information, use AES-CTR. I described a way to do it that was somewhat more efficient than just calling AES CTR DRBG.

3 If you need something that behaves more like a random oracle, use KMAC256. (It wouldn't be much harder to specify how to just use SHAKE for this.)

What else do we need for seed expansion/randomness?

--John

/////
Everyone,

We've been talking over some of the comments here over the last couple weeks, trying to figure out what useful guidance or code we can provide. Here's what I think we've understood:

a. Many PQC algorithms use a lot of (pseudo)random bits, enough that the performance of the source of random bits is important.

b. As Dan has pointed out, random numbers aren't free. In fact, the way I'd expect an implementation to work in the field is to call the system's or cryptographic library's cryptographically-strong RNG for each output.

c. If we assume the RNG is doing a NIST DRBG, then this imposes some extra costs on the algorithm. Our DRBGS (and anything else that meets the same security requirements) do an extra step at the end of each request for random bytes, where they guarantee "backtracking resistance"—that is, they guarantee that someone who compromises the RNG state in the future can't learn previous RNG outputs. And that's going to be true in the field, too—when you keep making calls to your crypto library's "get random bytes" call, it's going to do this extra cryptographic work to ensure backtracking resistance.

d. The obvious way around this is to request all the bytes you need up front—if you're

going to need a million bytes of RNG output, just call the RNG once and ask for a million bytes. For AES CTR DRBG (probably the fastest DRBG on most platforms), that will give you outputs at about the speed of AES in counter mode—the small extra overhead of guaranteeing backtracking resistance (aka rekeying AES a few times and generating a few extra bytes) won't have much impact.

e. My understanding is that some algorithms don't know how many random bytes they'll need when they make their initial RNG request. (I'm not sure how this affects timing side channels.) But it would be convenient to have a scheme that would let you generate only the random bytes you need, rather than making multiple calls to the RNG, or making a call for more bytes than you need.

f. Nigel pointed out that there are places where what's needed is a kind of "seed expander" algorithm, which will take an input seed and a diversifier (to allow multiple random streams from the same seed) and generate as many bytes as are needed. He also pointed out some other issues with output length and such.

For (a-e), and maybe for some cases of (f), it seems like the best solution is:

1 First, to request random bytes from a NIST DRBG, since that probably mirrors the performance they'll get making RNG calls in the field. We'll provide some code to the testing platforms to support this, but it's really just going to be AES CTR DRBG.

2 Second, to use AES in counter mode to expand a random seed to a longer random stream. I think it's important for us to explicitly say that this is okay. In fact, we'll provide some code for it. Here's how we think it should be done:

S = the seed, 32 random bytes
D = the diversifier, 4 bytes
L = the maximum output length that will be requested, a 32-bit unsigned integer.

K = S
C = D || L || 0^{64}

Then just run AES in counter mode from there. This can be used to generate output bytes where you don't know how many you'll need until the end.

The thing to notice here is that AES CTR will do a fine job of taking a random unknown-to-the-attacker seed and expanding it to a pseudorandom stream that's also unknown to the attacker. (The security here is just based on running AES in counter mode—

we're not inventing some new thing here!)

However, this won't work as a random oracle. That is, if you want to take a value that might be partly chosen by the attacker and use it to generate outputs that are outside the attacker's control, you probably don't get what you want from this scheme. (For example, if I get to choose the inputs, I can force the first 128 bits of output to a value of my choosing with 2^{64} work. And if you try to sketch out an indifferentiability proof for this scheme based on an ideal cipher model, you quickly run into the fact that your simulator can't handle decryption queries at all efficiently.)

Also, this probably makes sense to use only for cases where you don't know how many random bytes you'll need when you start generating them. If you know you'll need 100,000 output bytes, you can get that directly by calling the RNG (which will end up making two generate calls for AES CTR DRBG). The extra overhead in the DRBG calls will be a very small fraction of the total work to generate 6,250 output blocks from AES.

3 It's probably a good idea to make this seed-expanding mechanism its own thing inside the code, so that it can be changed out if there's a better technique available later.

4 If you need something that works more like a random oracle, we have a standard that I think will do the job nicely now: KMAC from SP 800-185. KMAC is defined in a way that allows both XOF-mode access (where you don't know the output length till you make the final call) and normal access where you request N bits of output and then you're done.

The general way to do this would be:

S = seed
D = diversifier (so you can have many streams from the same seed)
L = requested length

output = KMAC256(key=S, data = D, output_length = L)

KMAC is also defined for XOF mode, so you can feed in a seed and a diversifier and then keep requesting output bits until you don't need anymore.

KMAC256 is designed to have a 256-bit security level, and keeps an internal state of 512 bits throughout its computation, so I think it will support any security level we're targeting with the PQC algorithms.

Comments?

--John