From: Moody, Dustin (Fed)
To: Kelsey, John M. (Fed)
Subject: RE: Proposals (draft; please comment!)
Date: Wednesday, August 2, 2017 9:50:00 AM

Ok – thanks! I don't want you to worry about work on vacation, but anything you can do would be great. We want to send something to Dan and OpenQuantumSafe pretty soon.

From: Kelsey, John M. (Fed)
Sent: Wednesday, August 02, 2017 9:48 AM
To: Moody, Dustin (Fed) <dustin.moody@nist.gov>
Subject: Re: Proposals (draft; please comment!)

Dustin,

Let me think a bit more, ask Nicky about his concerns, and write the xof mode pseudocode. I should have some time to think about this later this week.

--John

On: 31 July 2017 11:54, "Moody, Dustin (Fed)" <dustin.moody@nist.gov> wrote:

Larry,

Does this have everything you need? John, any changes?

Dustin

From: Kelsey, John M. (Fed)
Sent: Thursday, July 27, 2017 8:09 PM
To: internal-pqc <internal-pqc@nist.gov>
Cc: McKay, Kerry A. (Fed) <kerry.mckay@nist.gov>; Sonmez Turan, Meltem (Assoc) <meltem.turan@nist.gov>; Mouha, Nicky W. (IntlAssoc) <nicky.mouha@nist.gov>
Subject: Proposals (draft; please comment!)

Here's what I think we want to say here: (This is a draft of what I'm planning to send Larry.)

(I'm CCing this to Kerry, Meltem, and Nicky because they're all symmetric crypto people who may see a flaw I'm missing.)

1 Seed Expanders

Some submitters have asked for a generic mechanism for taking a short seed and expanding it to a much longer seed, with various bells and whistles. Below, we specify two straightforward ways to do this which are based on NIST-approved cryptographic algorithms, and are closely related to existing key derivation functions and DRBGs. These aren't the only acceptable ways to do this, but they are

reasonable ones we believe are secure and sensible designs.

1.1 AES-based seed expander

Requirements: We need a starting seed of 32 bytes which is secret. We also need to choose a diversifier of 8 bytes. (Previously we had a four-byte diversifier, but some people wanted a longer one, and this has the same security properties.)

Use: This function is used to take 32-byte secret value and expand it to a long sequence of pseudorandom bits. If the secret value is a different size than 32 bytes, it must be processed with a hash function to turn it into a 32 byte secret value.

Inputs: S = the seed (32 bytes), D = the diversifier (8 bytes), L = the maximum length in bytes (4 bytes). L is represented as an unsigned integer in network byte order.

Outputs: Z = the expanded string, L bytes long.

encode(i) encodes a 32-bit unsigned integer as a four byte string in network byte order.

Algorithm:

AES_Expand_Seed(S, D, L)

```
K = S
V = D || encode(L)
t = L/16
Z = ""
for i = 0 to t-1:
        Z = Z || AES(K,V || encode(i)
return leftmost L bytes of Z
```

Security Claims:

If S is secret, then this will give a cryptographically strong pseudorandom sequence.

The output stream is domain-separated for different choices of D and L. That is, the sequence from two different choices of D, or two different choices of L, are basically unrelated.

The best attack I know of is a distinguishing attack: Suppose I want to see whether I'm looking at an ideal random sequence or the outputs from this sequence. I request 2^{32} bytes, which is 2^{28} AES blocks. If I am looking at a truly random sequence, the probability that at least two blocks collide is about 2^{55} 2^{-128} = 2^{-73}; when looking at the AES stream, collisions never happen.

In the best case, a distinguisher would have a 1/2 probability of distinguishing between the ideal and real stream. Because we're using a block cipher here, the distinguisher has a 1/2 + 2^{-73}

probability of distinguishing between the ideal and real stream.

Another way of thinking about this property: Anytime I see one AES output block in the sequence, I know that block will never recur. The probability that this particular block would recur somewhere else was very low (2^{-100}).

A similar insight applies to the domain separation here. If this were an ideal object of its kind, then the stream resulting from (S,D,L) and the one resulting from (S, D', L) would have absolutely no relationship. However, this object does have a very weak relationship—the output stream from (S,D,2^{32}) and (S, D', 2^{32}) will never have any pair of AES block outputs equal between them.

This is a way to get more data for the distinguishing attack, above. If we request 2^{32} diversifiers, and request 2^{28} AES blocks from each under the same value of S, we have a total of 2^{56} AES blocks. There will never be a collision in any 128-bit AES output block in these sequences; an ideal object of this kind would give output sequences with a probability of about 2^{-17} of having such a collision.

This would allow a distinguisher which had probability 1/2 + 2^{-17} of successfully distinguishing this seed expander from an ideal object of its kind.

[[Note: This analysis makes me want to decrease D to 32 bits to bound the attack, and maybe to limit L to a smaller value as well.]]

This is a pretty good way to get random bits, but it's not quite like a random oracle. One way in which it is different is this: Suppose I want to choose the inputs to force the first 128 bits to be all zeros, where we don't care about the rest of the string. If this behaved like a random oracle, I'd expect that effort to take me 2^{128} tries. However, with this function, it requires about 2^{32} AES decryptions:

a. Choose an AES key K.
b. Compute X = AES^{-1}(K, 0)
c. If the last 32 bits of X are all zeros, then I can construct an input that forces the first 128 bits to zero:

S = K.
D = the leftmost 8 bytes of X.
L = the next 4 bytes of X.

This will work as long as L >= 16, which happens with probability about 1-2^{-28}.

There may be many other ways this fails to be a random oracle that I haven't noticed. (Certainly if you try to do an indifferentiability proof, you'll run into the problem that the simulator for decryption has to be very inefficient.)

1.2 KMAC-based seed expander

KMAC 256 is a keyed hashing construction based on Keccak that appears in SP 800-185.

KMAC256 takes as inputs a key K, an input string X, a length L, and an optional customization string S. We can write it as KMAC(K,X,L,S). It produces an L-bit string as a result.

The KMAC seed expander takes the parameters S = the seed, D = the distinguisher, and L = the length.

Requirements: We need an input seed (S) , a diversifier (D), and an output length in bytes (B).

Use: This function takes a seed, diversifier, and length, and produces a random-looking output string.

Outputs: An B-bit output string

Algorithm:

KMAC_Expand_Seed(S, D, B)

L = B*8
Z = KMAC256(S, D, L, “”)
return Z

Security Claims:

KMAC256 is built on top of Keccak with a 512-bit capacity, thus with a 256-bit security level against all attacks, so long as the output is at least 512 bits.

When S is secret, KMAC is a PRF.

When S is known or even chosen by the attacker, KMAC256 simply amounts to encoding the inputs (S,D,L) in an unambiguous way and feeding them to Keccak with a 512 bit capacity. We can use the existing indifferentiability proof to reason about the security of KMAC when used in this way, assuming that the underlying permutation is ideal.

Informally, we expect the KMAC_Expand_Seed() function to behave just as we’d expect a good hash function to behave. In particular, there should be no shortcuts to forcing any given bits of the output sequence to a specific output value or otherwise exerting any control on those outputs, short of brute-force computations.

In general, the KMAC seed expander is safer to use when the seed isn’t secret, but the AES seed expander is very likely to be faster.

Again, none of these are ***required***, they are simply things we suggest as reasonable choices.

2 Randomness

[[This is where I'd really like comments]]

We want to get a sense of how fast PQC algorithms will be in practice, using NIST-approved DRBGs. We recognize that there are faster ways to generate pseudorandom bits which may be appropriate in some situations—for example, not every algorithm may even need cryptographic pseudorandomness. However, when we consider how PQC algorithms will actually be implemented in FIPS-compliant systems, they will be ultimately making calls to a NIST approved DRBG. In terms of performance, the fastest DRBGs from SP 800-90A that meet a 256-bit security level requirement will be AES256_CTR_DRBG, SHA256_Hash_DRBG, or SHA512_Hash_DRBG, depending on the platform.

Some people have raised concerns about a potential source of inefficiency here: Each call to one of these DRBGs involves a step to ensure forward secrecy (backtracking resistance, in 90A's language). That is, we do a little extra work to make sure that even if the DRBG were compromised in the future, it would not reveal past random bits. In most situations, this doesn't cause any noticeable problem. However, some PQC algorithms use a lot of random bits. A naive implementation of a PQC algorithm that needed 100 128-bit random numbers might simply call the DRBG requesting 128 bits for each value, 100 separate calls, and thus pay a pretty big performance penalty. (The fastest possible way of generating 100 128-bit values from AES_CTR_DRBG would require 103 AES encryptions; the naive way would require 400 AES encryptions. An extra 300 AES encryptions isn't all ***that*** much time, but it definitely might make a difference in benchmarking algorithms.)

It's possible to change the RNG available for performance testing the candidate PQC algorithms to get rid of this performance impact; Dan's proposal for buffering outputs is one way. However, this leaves a potential problem—assuming the PQC algorithms are implemented on FIPS-compliant systems, or any other system where each PRNG call they make involves some kind of backtracking resistance step, they may find themselves getting lousy performance.

Our proposed solution to this is inspired by Dan's recent proposal—the implementation needs to use a NIST DRBG to generate random bits (so we get accurate performance numbers), but should buffer them if they expect to be making a large number of calls to the DRBG. This isn't really worthwhile if the algorithm isn't going to make a lot of calls to the DRBG.

An algorithm which is very likely to request a total of no more than N RNG bytes can then do the following:

```
unsigned char buffer[N];
...

D = AES256_CTR_DRBG(get_random(32))
buffer = generate(D,N)
```

Then, the bytes are read from the buffer as they're needed. If more bytes are needed than are in the buffer, then the DRBG may be called again to refill it.

This is almost exactly equal to Dan's proposal before. The differences are:

a. The buffering is only done by the implementation if it is needed.

[[I'm assuming anyone capable of implementing a novel cryptographic algorithm can pretty easily handle writing the buffering code, but we could write some code and make it available if we need to.]]

b. The implementation chooses the size of the buffer based on their expected need for random bytes. An algorithm that needs only 8 bytes of randomness will never pay for 768 bytes.

c. The resulting implementations will be able to give comparable performance when implemented on systems whose PRNGs do some extra work on each randomness request.

Comments?

I still need to write something back to Dan, and I've cooled off enough to probably do it. On the other hand, I'm juggling about fifty things getting ready to go on vacation, so my stress level (and background annoyance level) is probably higher than normal. I will be online tomorrow (assuming all goes well—I'm going to end up at Dulles for some of the day and will likely have to wait around there for a few hours). But I'd like to turn this around by the end of the day tomorrow, so please let me know what you think ASAP.

Thanks,

--John