Q: Does the requirement for ANSI C source code preclude the use of assembly language optimizations?  
   This question is still part of the FAQ. The question and answer have been modified.

A: The optimized code required as part of the submission package should be ANSI C with no assembly (this includes inline assembly). This code is meant to be portable. If significant optimizations can be made with assembly, then it can be included as an additional implementation and discussed in the performance analysis.

Q: Can third party open-source code be used in submissions?  
   This question is still part of the FAQ. The answer has been modified.

A: In both the mandatory reference implementation and the mandatory optimized implementation, submissions may use NTL Version 10.5.0 (http://www.shoup.net/ntl/download.html), GMP Version 6.1.2 (https://gmplib.org), and OpenSSL Version 1.10f (https://www.openssl.org/source). Submitters may assume that these libraries are installed on the reference platform and do not need to provide them along with their submissions.

If a submitter wishes to use a third-party open source library other than the ones specified above, they must send a request to NIST at pqc-comments@nist.gov by September 1st, 2017, with the name of the library and a link to the primary website hosting it from which it may be downloaded. NIST will either approve or deny this request within 2 weeks of receiving it. Should a request be approved, it will be added to the above list of acceptable third-party open source libraries provided in this FAQ.

In short, they may be used, with the following caveats.

1. The library source code should be integrated into the submission package in a self-contained manner. This means that the submission package, All submission packages using third-party open source code, should contain build scripts which will allow for seamless “one-stop” building of the submitter’s original code and all dependencies.

For example, on a Linux platform, it should require no more work to build the than running the standard succession of commands. In particular, the build process should not require the installation of any new libraries that are not contained in the submission package—we should be able to find the versions of these libraries specified above that will be pre-installed on the reference platform.
Separate build scripts should be included for the reference Windows platform and reference Linux platform that work using the GCC Compiler Collection version 6.4.0 (or ports thereof) and related tools as well as any platform-specific commands required.

In addition, as part of the written submission, the submitter shall describe in their own words the functionalities provided by any algorithms from third-party open-source libraries that are used in the implementations.

Q: How does a submission obtain secure randomness?
This question is still part of the FAQ, and was originally added 03/29/2017. The answer has been modified twice.

A: The function randombytes() will be available to the submitters. This is a function from the SUPERCOP test environment and should be used to generate seed values for an algorithm. Randombytes should only be used to seed a NIST-approved DRBG if the algorithm needs additional randomness beyond the seed value a NIST-approved DRBG should be used. As stated in the call for algorithms, the DRBG should be NIST-approved. If a non-approved DRBG is used “the submitter shall provide an explanation for why a NIST-approved primitive would not be suitable.” The length of the random value obtained from randombytes() should be selected to match one of the security categories in the call for algorithms. That is, if the call to generate a key pair is from category 1 the randomness value should be 192 bits (24 bytes), if the call is from category 2 or 3 it should be 256 bits (32 bytes) and if it is from category 4 or 5 it should be 320 bits (40 bytes). The DRBG will be used to expand that if necessary.

For functional and timing tests a deterministic generator is used inside randombytes() to produce the seed values. If security testing is being done simply substitute calls to a true hardware RBG inside randombytes().

Function prototype for randombytes() is:

```c
// The xlen parameter is in bytes
void randombytes(unsigned char *x, unsigned long long xlen)
```

The following demonstrate the use of the KAT and non-KAT versions of the functions to generate a key pair for encryption:

```c
int crypto_encrypt_keypair_KAT(  
    unsigned char *pk,  
    unsigned char *sk,  
    const unsigned char *randomness
)

int crypto_encrypt_keypair(unsigned char *pk, unsigned char *sk)  
{  
    unsigned char pk[CRYPTO_PUBLICKEYBYTES];
```
unsigned char sk[CRYPTO_SECRETKEYBYTES];
unsigned char seed[CRYPTO_RANDOMBYTES];

randombytes(seed, CRYPTO_RANDOMBYTES);
crypto_encrypt_keypair_KAT(pk, sk, seed);

Q: Why are hash functions assigned fewer bits of quantum security than classical security?

A: Bernstein\textsuperscript{1} is widely cited as demonstrating that the most efficient quantum algorithm for finding hash collisions is the classical algorithm given by Van Oorschot and Wiener\textsuperscript{2}. NIST believes this analysis is correct. Nonetheless, NIST's security goal, that schemes claiming s bits of quantum security be at least as secure against cryptanalysis as a 2s bit block cipher leads to differing definitions for quantum and classical security. In particular, quantum search for a 2s bit key does not parallelize well. It is NIST's judgement that, since cryptanalysis in the real world tends to be most successful when it can take advantage of highly parallel implementations for attacks, finding collisions in a 2s bit hash function must be considered easier than searching for the key of a 2s-bit block cipher, even in a world with ubiquitous quantum computing. NIST therefore assigns fewer than s bits of quantum security against collision to 2s bit hash functions.

Q: What is the rationale to convert time and space complexity of known attacks into a single number for quantum and classical security?

A: NIST's definition of s bits of quantum security is "as hard to break as a block cipher with a 2s bit key, assuming a relatively efficient and scalable quantum computing architecture is available." According to the analysis of Zalka\textsuperscript{3} the best generic quantum attack on a 2s-bit block cipher requires a quantum circuit with depth*(squeareroot (space)) proportional 2^s. This would suggest that quantum security should be defined as the minimum possible value of log(depth*(squeareroot (space))) plus a constant (to put the quantum security of AES 128 at precisely 64 bits of quantum security,) accross all quantum and classical algorithms. This formula should only be taken as a rough guess, though, as there are additional factors to consider: Extremely serial and extremely parallel attacks are likely to be of limited practical relevance, even if the above formula rates them as most efficient. Likewise, even under the assumption that a relatively scalable and efficient quantum computing architecture is available, it is still likely that purely classical algorithms will be easier to implement than the formula suggests, and quantum algorithms that, unlike parallel versions of Grover's algorithms, cannot be divided into small, unentangled,

\textsuperscript{1} Daniel J. Bernstein, Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete? https://cr.yp.to/hash/collisioncost-20090517.pdf
subcircuits, will be harder to implement than the formula suggests. NIST plans to take these practical considerations into account when making its evaluations.

Similarly, NIST’s definition of s bits of classical security is “as hard to break as a block cipher with an s bit key, assuming quantum computers are not available.” This suggests that classical security should be estimated as the minimum value of \( \log(\text{depth} \times \text{space}) \) plus a constant, over all classical attack algorithms.