PQC - API notes

Most of the API information is derived from the **eBATS: ECRYPT Benchmarking of Asymmetric Systems** (https://bench.cr.yp.to/ebats.html). This has been done to facilitate benchmarking algorithm performance. Please look at the eBATS page for more information on how to submit an algorithm for performance benchmarking.

Your functions must have exactly the prototypes shown here. For example, the `crypto_sign_keypair` function must have an `unsigned char` pointer for the public-key output and then an `unsigned char` pointer for the secret-key output. Your functions must return 0 to indicate success, -1 to indicate an error condition (other negative numbers may be used to indicate specific failures, e.g., out of memory).

Public-key Signatures

See https://bench.cr.yp.to/call-sign.html for more information on Public-key Signature API and performance testing.

The first thing to do is to create a file called *api.h*. This file contains the following three lines (with the sizes set to the appropriate values):

```
#define CRYPTO_SECRETKEYBYTES 256
#define CRYPTO_PUBLICKEYBYTES 85
#define CRYPTO_BYTES 128
```

indicating that your software uses a 256-byte (2048-bit) secret key, an 85-byte (680-bit) public key, and *at most* 128 bytes of overhead in a signed message compared to the original message.

Then create a file called *sign.c* with the following function calls:

Generates a keypair - *pk* is the public key and *sk* is the secret key.

```
int crypto_sign_keypair(
     unsigned char *pk,
     unsigned char *sk
)
```

Sign a message: *sm* is the signed message, *m* is the original message, and *sk* is the secret key.

```
int crypto_sign(
     unsigned char *sm, unsigned long long *smlen,
     const unsigned char *m, unsigned long long mlen,
     const unsigned char *sk
   )
```

Verify a message signature: *m* is the original message, *sm* is the signed message, *pk* is the public key.

```
int crypto_sign_open(
     unsigned char *m, unsigned long long *mlen,
     const unsigned char *sm, unsigned long long smlen,
     const unsigned char *pk
   )
```

Public-key Encryption

See https://bench.cr.yp.to/call-encrypt.html for more information on Public-key Encryption API and performance testing.

The first thing to do is to create a file called *api.h*. This file contains the following three lines (with the sizes set to the appropriate values):

```
#define CRYPTO_SECRETKEYBYTES 256
#define CRYPTO_PUBLICKEYBYTES 64
#define CRYPTO_BYTES 48
```

indicating that your software uses a 256-byte (2048-bit) secret key, a 64-byte (512-bit) public key, and *at most* 48 bytes of overhead in an encrypted message compared to the original message.

Then create a file called *encrypt.c* with the following function calls:

Generates a keypair - *pk* is the public key and *sk* is the secret key.

```
int crypto_encrypt_keypair(
     unsigned char *pk,
     unsigned char *sk
)
```

Encrypt a plaintext: *c* is the ciphertext, *m* is the plaintext, and *pk* is the public key.

```
int crypto_encrypt(
     unsigned char *c, unsigned long long *clen,
     const unsigned char *m, unsigned long long mlen,
     const unsigned char *pk
   )
```

Decrypt a ciphertext: *m* is the plaintext, *c* is the ciphertext, and *sk* is the secret key.

```
int crypto_encrypt_open(
     unsigned char *m, unsigned long long *mlen,
     const unsigned char *c, unsigned long long clen,
     const unsigned char *sk
   )
```

Key Encapsulation Mechanism (KEM)

The calls in the eBATS specification do not meet the calls specified in the call for algorithms. However, attempts were made to match the specifications for the other algorithms.

The first thing to do is to create a file called *api.h*. This file contains the following four lines (with the sizes set to the appropriate values):

```
#define CRYPTO_SECRETKEYBYTES 192
#define CRYPTO_PUBLICKEYBYTES 64
#define CRYPTO_BYTES 64
#define CRYPTO_CIPHERTEXTBYTES 128
```

indicating that your software uses a 192-byte (1536-bit) secret key, a 64-byte (512-bit) public key, a 64-byte (512-bit) shared secret, and at most a 128-byte (1024-bit) ciphertext.

Then create a file called *kem.c* with the following function calls:

Generates a keypair - *pk* is the public key and *sk* is the secret key.

```
int crypto_kem_keypair(
     unsigned char *pk,
     unsigned char *sk
)
```

Encrypt - *pk* is the public key, *ct* is a key encapsulation message (ciphertext), *ss* is the shared secret.

```
int crypto_kem_enc(
     unsigned char *ct,
     unsigned char *ss,
     const unsigned char *pk
)
```

Decrypt - *ct* is a key encapsulation message (ciphertext), *sk* is the private key, *ss* is the shared secret

```
int crypto_kem_dec(
     unsigned char *ss,
     const unsigned char *ct,
     const unsigned char *sk
)
```

Additional functions

A function, *randombytes()*, will be available to obtain random input. For Known Answer Tests (KAT), and on the NIST Reference Platforms, this function is AES_CTR_DRBG (see SP800-90A section 10.2.1.5.1). The function prototype comes from the SUPERCOP package (https://bench.cr.yp.to/supercop.html). The type for the length

argument is more than needed, but is left for consistency with the SUPERCOP package. The calling function shall allocate the storage for *x* and the *xlen* parameter specifies a number of bytes.

```
void randombytes(unsigned char *x,
                 unsigned long long xlen)
```

To facilitate Known Answer Tests, a function randombytes_init() is provided to deterministically instantiate AES_CTR_DRBG (see SP 800-90A section 10.2.1.3.1). The inputs are *entropy_input*, *personalization_string*, and *security_strength*. The *security_strength* input shall be set to 256, the *personalization_string* may be omitted passing a NULL pointer. The length of *entropy_input* shall be fixed at 384 bits (48 bytes). This function is only called by the test code to verify KAT values.

```
void randombytes_init(unsigned char *entropy_input,
             unsigned char *personalization_string,
             int security_strength)
```

A function, *seedexpander()*, will be available to generate additional pseudorandom material. The calling function shall allocate the storage for *x* and the *xlen* parameter specifies a number of bytes. This function is used to generate data of arbitrary length with the additional feature that two calls for 8 bytes will produce the same data as a single call for 16 bytes.

```
void seedexpander(AES_XOF_struct *ctx,
                  unsigned char *x,
                  unsigned long xlen)
```

A function, *seedexpander_init()*, will be available to initialize the *seedexpander()* function. Input values are the *seed* (a 32 byte value), a *diversifier* (an 8 byte value), and a *max_length* (a value less 2**32). This function must be called whenever *seedexpander()* is used.

```
void seedexpander_init(AES_XOF_struct *ctx,
                       unsigned char *seed,
```

```
                                unsigned char *diversifier,
                                unsigned long maxlength)
```

The following structure is used to store the context of the seed expander so that multiple instances can exist concurrently.

```
typedef struct {
    unsigned char   buffer[16];
    int             buffer_pos;
    unsigned long   length_remaining;
    unsigned char   key[32];
    unsigned char   ctr[16];
} AES_XOF_struct;
```