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. There are two sets of API calls listed for each primitive. The first set is the API call directly from the eBATS page, or something very similar for the Key Encapsulation Mechanism section. The second set of calls is for testing purposes. The calls extend the eBATS calls for functions that utilize randomness by providing a pointer to specify a randomness string. This will allow algorithms that utilize randomness to be able to provide reproducible results. For example, this will allow testing of KAT files and other sample values.

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 four lines (with the sizes set to the appropriate values):

```
	#define CRYPTO_SECRETKEYBYTES 256
	#define CRYPTO_PUBLICKEYBYTES 85
	#define CRYPTO_BYTES 128
	#define CRYPTO_RANDOMBYTES 32
```

indicating that your software uses a 256-byte (2048-bit) secret key, an 85-byte (680-bit) public key, *at most* 128 bytes of overhead in a signed message compared to the original message, and 32 bytes of random input.

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

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

KAT calls

```
         int crypto_sign_keypair_KAT(
              unsigned char *pk,
              unsigned char *sk,
              const unsigned char *randomness
         )
         int crypto_sign_KAT(
              unsigned char *sm, unsigned long
long *smlen,
              const unsigned char *m, unsigned
long long mlen,
              const unsigned char *sk,
              const unsigned char *randomness
         )
```

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 four lines (with the sizes set to the appropriate values):

```
     #define CRYPTO_SECRETKEYBYTES 256
     #define CRYPTO_PUBLICKEYBYTES 64
     #define CRYPTO_BYTES 48
     #define CRYPTO_RANDOMBYTES 32
```

indicating that your software uses a 256-byte (2048-bit) secret key, a 64-byte (512-bit) public key, *at most* 48 bytes of overhead in an encrypted message compared to the original message, and 32 bytes of random input.

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

eBATS 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
         )
```

KAT calls

```
         int crypto_encrypt_keypair_KAT(
              unsigned char *pk,
              unsigned char *sk,
              const unsigned char *randomness
         )
         int crypto_encrypt_KAT(
              unsigned char *c, unsigned long
long *clen,
              const unsigned char *m, unsigned
long long mlen,
              const unsigned char *pk,
              const unsigned char *randomness
         )
```

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
     #define CRYPTO_RANDOMBYTES 32
```

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, at most a 128-byte (1024-bit) ciphertext, and 32 bytes of random input.

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

eBATS-like calls

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

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

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

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

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

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

KAT calls

```
int crypto_kem_keygenerate_KAT(
     unsigned char *pk,
     unsigned char *sk,
     const unsigned char *randomness
)
int crypto_kem_encapsulate_KAT(
     unsigned char *ct,
     unsigned char *ss,
     const unsigned char *pk,
     const unsigned char *randomness
)
```

Additional function

A function will be available to obtain random input. 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)
```

Sample code

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

```
         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);
         }
```