From:	Moody, Dustin (Fed)
To:	Alperin-Sheriff, Jacob (Fed); Bassham, Lawrence E. (Fed); Chen, Lily (Fed); Daniel Smith-Tone; Jordan, Stephen P (Fed); Liu, Yi-Kai (Fed); Miller, Carl A. (Fed); Moody, Dustin (Fed); Peralta, Rene C. (Fed); Perlner, Ray A. (Fed); Smith-Tone, Daniel C. (Fed)
Subject:	Proposed post to the pqc-forum on KEMs and the API
Date:	Wednesday, October 26, 2016 9:59:54 AM

Everyone,

We'd also like to get some feedback on our approach to using KEMs, as that was the other main area of comments we received. I've written a first attempt at such a post to be made on the pqc-forum (see below). Let me know what you think by the end of this week. We also plan to post the updated API, so we can let some of the more knowledgeable people in that field give us their input.

Thanks,

Dustin

NIST received several comments regarding our request for a key-exchange algorithm. As a result, we are clarifying what exactly we are looking for. In our revised call, instead of using the term key-exchange we will be asking for Key Encapsulation Mechanisms (KEMs). KEM schemes consist of algorithms for key generation, encapsulation, and decapsulation.

One important application is using public-key cryptography to securely establish a key to be used for symmetric encryption. In SP 800-57, *Recommendation for Key Management*, NIST has distinguished between two methods of achieving this functionality. The first uses public-key encryption algorithms, which are referred to as key-transport in SP 800-57, while the second uses KEMs, which are referred to as key agreement or key-exchange. NIST intends to standardize one or more schemes that enable semantically secure encryption or key encapsulation with respect to adaptive chosen ciphertext attack (IND-CCA2), for general use.

While chosen ciphertext security is necessary for many existing applications, it is possible to implement a purely ephemeral key exchange protocol in such a way that only passive security is required from the encryption or KEM primitive. For these applications, NIST will consider standardizing an encryption or KEM scheme which provides semantic security with respect to chosen plaintext attack (IND-CPA).

We note that KEM and public-key encryption functionalities can generally be interconverted. Unless otherwise specified by the submitter, NIST will apply standard conversion techniques to public-key encryption algorithms, so that the resulting scheme can be considered for standardization as a KEM, and vice versa.

We would like your feedback. Does this approach seem sound? What (if any) changes would you suggest.

The updated API:

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 64
```

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 64 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 64
```
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 64 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 64
```
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 64 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(
const unsigned char *pk,
unsigned char *ct,
unsigned char *ss
)
```

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

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

KAT calls

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