

OPEN QUANTUM SAFE LIBRARY : LIBOQS

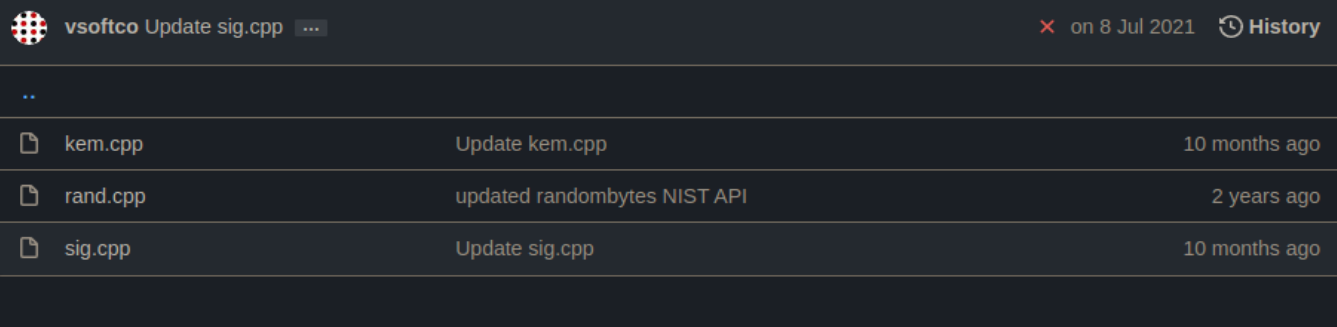
- By Aditya Koranga

WHAT IS LIBOQS?

Liboqs is an open-source C library for quantum-safe cryptographic algorithms. It contains a collection of open-source implementation of quantum-safe key encapsulation mechanism (KEM) and digital signature algorithms such as Kyber, Saber, NTRU , McEliece, Frodo , Dilithium, RainBow, Picnic, etc. For knowing how implementation of such algorithms works let's see the cpp wrapper of liboqs. <https://github.com/open-quantum-safe/liboqs-cpp> .

LIBOQS-CPP

Liboqs-cpp offers a C++ wrapper for the Open Quantum Safe liboqs C library. The wrapper is written in standard C++ 11. So the complete library contains just 3 important files: kem.cpp, rand.cpp and sig.cpp. If you are able to understand these three files then you can easily understand the whole implementation.



| | | | |
|----------------------------|------------------------------|-----------------|---------|
| vsoftco Update sig.cpp ... | | ✗ on 8 Jul 2021 | History |
| .. | | | |
| 📄 kem.cpp | Update kem.cpp | 10 months ago | |
| 📄 rand.cpp | updated randombytes NIST API | 2 years ago | |
| 📄 sig.cpp | Update sig.cpp | 10 months ago | |

Let's understand these files one by one.

KEM.cpp

Link:

<https://github.com/open-quantum-safe/liboqs-cpp/blob/main/examples/kem.cpp>

- Firstly we will select the algorithm that we are going to use. Here in this, we have taken 'Kyber512' and will get the related details.

```
14
15     std::string kem_name = "Kyber512";
16     oqs::KeyEncapsulation client{kem_name};
17     std::cout << "\n\nKEM details:\n" << client.get_details();
18
```

- Now we will need the public key of the client for which there is a key generation process.

```
20     oqs::bytes client_public_key = client.generate_keypair();
21     t.toc();
22     std::cout << "\n\nClient public key:\n" << oqs::hex_chop(client_public_key);
23     std::cout << "\n\nIt took " << t << " millisecs to generate the key pair";
24
```

- Now with help of this public key of the client, the server will encapsulate the message and there will be two things that will get as output: ciphertext and shared secret.

```
25     oqs::KeyEncapsulation server{kem_name};
26     oqs::bytes ciphertext, shared_secret_server;
27     t.tic();
28     std::tie(ciphertext, shared_secret_server) =
29         server.encap_secret(client_public_key);
```

- Now with this ciphertext we will start the decapsulation process (the shared secret will remain secret i.e. known only to the server). Now the client will decapsulate this ciphertext with the help of his secret key (that will be known to the client only) and the output of this process will also be a shared secret.

```
34     oqs::bytes shared_secret_client = client.decaps_secret(ciphertext);
35     t.toc();
36     std::cout << "\nIt took " << t << " millisecs to decapsulate the secret";
37
38     std::cout << "\n\nClient shared secret:\n"
```

- Now, the final step. If the shared secret that we got after the encapsulation process (i.e. in the third step) and the shared secret that we got after the decapsulation process (i.e. in the fourth step) are equal to each other then it is a 'valid' process and the client is able to get the message safely otherwise it will show an error.

```
42     bool is_valid = (shared_secret_client == shared_secret_server);
43     std::cout << "\n\nShared secrets coincide? " << is_valid << '\n';
44
45     return is_valid ? oqs::OQS_STATUS::OQS_SUCCESS : oqs::OQS_STATUS::OQS_ERROR;
46 }
```

RAND.cpp

Link:

<https://github.com/open-quantum-safe/liboqs-cpp/blob/main/examples/rand.cpp>

This file is basically for the random number generation process. Random numbers are also an important thing to keep in mind. More a number is random more will be its security. This random number will be able for various important things such as the formation of key pairs.

- For the formation of random numbers we need an entropy seed, as the name suggests, this seed will be the initial step of a random number which will then pass through various arithmetic operations to form a big random number. In this liboqs library, the length of this entropy seed is taken 48 bytes.

```
oqs::rand::randombytes_switch_algorithm(OQS RAND_alg_nist_kat);
oqs::bytes entropy_seed(48);
entropy_seed[0] = 100;
entropy_seed[20] = 200;
entropy_seed[47] = 150;
oqs::rand::randombytes_nist_kat_init_256bit(entropy_seed);
std::cout << std::setw(18) << std::left;
std::cout << "NIST-KAT: " << oqs::rand::randombytes(32) << '\n';
```

- Now there are different methods of random number generators, custom RNG, we can use OpenSSL for generating it,

```

27     oqs::rand::randombytes_custom_algorithm(custom_RNG);
28     std::cout << std::setw(18) << std::left;
29     std::cout << "Custom RNG: " << oqs::rand::randombytes(32) << '\n';
30
31 // we do not yet support OpenSSL under Windows
32 #ifndef _WIN32
33     oqs::rand::randombytes_switch_algorithm(OQS RAND_alg_openssl);
34     std::cout << std::setw(18) << std::left;
35     std::cout << "OpenSSL: " << oqs::rand::randombytes(32) << '\n';

```

- You can also switch the algorithm for generating random .

```

38     oqs::rand::randombytes_switch_algorithm(OQS RAND_alg_system);
39     std::cout << std::setw(18) << std::left;
40     std::cout << "System (default): " << oqs::rand::randombytes(32) << '\n';

```

Note: The method that we use currently for generating random numbers is not considered to a true random number generation method as it also uses its own algorithm for generating the randomness. Currently liboqs also uses DRBG which is not a true random bit generator. The security will be really high when we will use true random numbers.

SIG.cpp

Link:

<https://github.com/open-quantum-safe/liboqs-cpp/blob/main/examples/sig.cpp>

This is the final step of this liboqs library. This step is for the verification that the sender is valid and not some unknown party.

- Firstly we will select the signature algorithm and take the message that is to be signed and in this file, we have taken 'Diltithium2' signature algorithm.

```

14     oqs::bytes message = "This is the message to sign"_bytes;
15     std::string sig_name = "Dilithium2";
16     oqs::Signature signer{sig_name};
17     std::cout << "\n\nSignature details:\n" << signer.get_details();

```

- Then we will generate the public key of the signer which will be used for later verification

```

20     oqs::bytes signer_public_key = signer.generate_keypair();
21     t.toc();
22     std::cout << "\n\nSigner public key:\n" << oqs::hex_chop(signer_public_key);
23     std::cout << "\n\nIt took " << t << " microsecs to generate the key pair";

```

- Now the signer will sign the message with his signing key(secret key) that will be known to him only.

```

26     oqs::bytes signature = signer.sign(message);
27     t.toc();
28     std::cout << "\nIt took " << t << " microsecs to sign the message";
29     std::cout << "\n\nSignature:\n" << oqs::hex_chop(signature);
30

```

- Now the final step of verification. We will now verify the signature with the help of the corresponding public key that we generated in the second step. As both the keys were of the same key pair, therefore the verification can be done in such a way.

```

31     oqs::Signature verifier{sig_name};
32     bool is_valid = verifier.verify(message, signature, signer_public_key);
33     std::cout << "\n\nValid signature? " << is_valid << '\n';
34
35     return is_valid ? oqs::OQS_STATUS::OQS_SUCCESS : oqs::OQS_STATUS::OQS_ERROR;

```

So, these were all the steps that are used in this library. We can even use different KEM and signature algorithms other than Kyber and Dilithium (both belong to the lattice family) that are considered safe by NIST. So this was the complete flow of this library,