Beyond Portable C: Rethinking PQC Implementations with Modern C++ Patterns

RWPQC 2026
Taipei City, Taiwan

Anjan Roy

Software Engineer with deep knack for Applied Cryptography

Ex-Polygon Labs, Ex-TII

Engineering @ Avail Project + PhD @ Radboud University

An official website of the United States government Here's how you know

NEWS

NIST Asks Public to Help Future-Proof Electronic Information

December 20, 2016

The National Institute of Standards and Technology (NIST) is officially asking the public for help heading off a looming threat to information security: quantum computers, which could potentially break the encryption codes used to protect privacy in digital systems. NIST is requesting methods and strategies from the world's cryptographers, with the deadline less than a year away.

The Call for Proposals for Post-Quantum Cryptography Standardization, announced today in the *Federal Register*, is NIST's first formal step toward countering the danger that quantum computers pose to the security of digital information. Though practical quantum computers have yet to be built, their design—which would draw upon very different scientific concepts than conventional computers—would enable them to break some of the cryptographic algorithms commonly used to protect electronic messages

MEDIA CONTACT

Chad Boutin
charles.boutin@nist.gov
(301) 975-4261

ORGANIZATIONS

An official website of the United States government Here's how you know

NEWS

NIST Announces First Four Quantum-Resistant Cryptographic Algorithms

Federal agency reveals the first group of winners from its six-year competition.

July 5, 2022



MEDIA CONTACT

Chad Boutin

charles.boutin@nist.gov

(301) 975-4261

An official website of the United States government Here's how you know

NEWS

NIST Releases First 3 Finalized Post-Quantum Encryption Standards

August 13, 2024

- NIST has released a final set of encryption tools designed to withstand the attack of a quantum computer.
- These post-quantum encryption standards secure a wide range of electronic information, from confidential email messages to e-commerce transactions that propel the modern economy.
- NIST is encouraging computer system administrators to begin transitioning to the new standards as soon as possible.

MEDIA CONTACT

Chad Boutin
charles.boutin@nist.gov
(301) 975-4261

```
Author: Anjan Roy <hello@itzmeanjan.in>
Date:   Mon Sep 19 11:55:39 2022 +0400

    deterministic uniform sampling in R_q | q = 3329

    Signed-off-by: Anjan Roy <hello@itzmeanjan.in>

added: include/sampling.hpp
```

• 1:

2nd commit of ml-kem C++ implementation

```cpp
#pragma once
#include "shake128.hpp"
#include <cstdint>

// IND-CPA-secure Public Key Encryption Scheme
namespace indcpa {

// Uniform sampling in R_q | q = 3329
//
// Given a byte stream, this routine *deterministically* samples a degree 255
// polynomial in NTT representation. If the byte stream is statistically close
// to uniform random byte stream, produced polynomial coefficients are also
// statiscally close to randomly sampled elements of R_q.
//
// See algorithm 1, defined in Kyber specification, present in NIST PQC final
// round submission package
// https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/submissions/Kyber-Round3.zip
static void
parse(shake128::shake128* const __restrict hasher, // Squeezes arbitrary bytes
      uint16_t* const __restrict poly               // Degree 255 polynomial
)
{
  constexpr size_t n = 256;
  constexpr uint16_t q = ((1 << 8) * 13) + 1;

  size_t i = 0;
  uint8_t buf[3]{};

  while (i < n) {
    hasher->read(buf, sizeof(buf));

    const uint16_t d1 = (static_cast<uint16_t>(buf[1] & 15) << 8) ^
                        (static_cast<uint16_t>(buf[0]) << 0);
    const uint16_t d2 = (static_cast<uint16_t>(buf[1] >> 4) << 0) ^
                        (static_cast<uint16_t>(buf[2]) << 4);

    if (d1 < q) {
      poly[i] = d1;
      i++;
    }

    if ((d2 < q) && (i < n)) {
      poly[i] = d2;
      i++;
    }
  }
}

}
```

```
ml-kem on  master [?] via △ v3.31.6
```

Yet Another Implementation of NIST PQC Selected Candidates

1,168 views

Subscribe



Anjan Roy
to pqc-forum

Feb 18, 2023, 5:13:50 PM

Hi NIST PQC community,

Hope you're doing well.

I'm writing to you for informing you about an initiative I took few months ago, which has recently come to an end. I decided to implement all NIST selected PQC candidates as minimal, header-only, zero-dependency, easy-to-use C++ library. I've implemented Kyber KEM (and PKE), Dilithium, SPHINCS+ and Falcon DSA.

> From above ordering you might have already guessed why Falcon is listed at the end 😉. It took me sometime to figure.

Below I attach few details regarding implementation of these four selected candidates. In their respective repository, you may find how to test/ benchmark/ use these libraries.

Before moving to next part, I'd like to mention that though my Falcon implementation is functionally correct **it is not yet tested to be conformant** with Known Answer Tests found in NIST submission. One shoutout, during Falcon implementation I found Thomas Prest's https://github.com/tprest/falcon.py very helpful along with the specification and reference implementation. Other three implementations (i.e. Kyber, Dilithium and SPHINCS+) **are tested with Known Answer Tests** (found in respective NIST submission packages) for conformance and correctness.

Following benchmark results were collected on an Intel Core i5-8279U CPU @ 2.40GHz Macbook Pro, using google-benchmark (more @ https://github.com/google/benchmark) as harness and Clang as C++ compiler. You'll find more benchmark results in respective repository.

Pulse
Contributors
Community
Community standards
Traffic
Commits
Code frequency
Dependency graph
Network
Forks
Actions usage metrics
Actions performance metrics

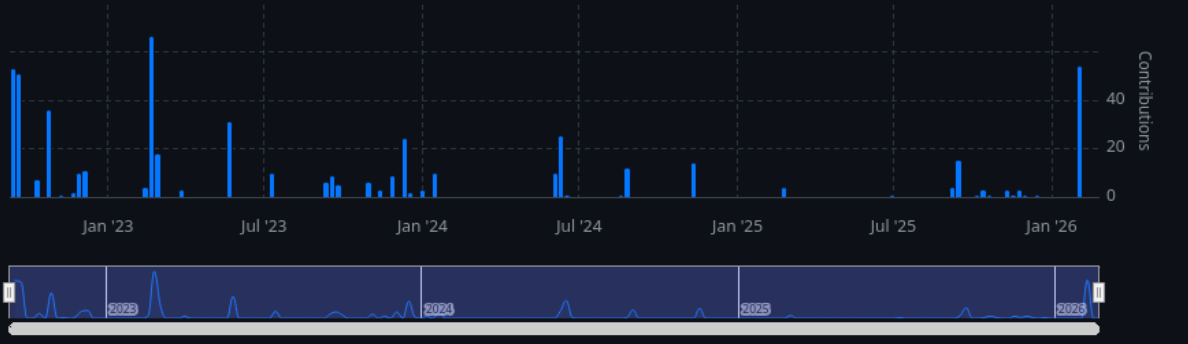
Contributors

Contributions per week to master, excluding merge commits

Period: All | Contributions: Commits

Commits over time

Weekly from 18 Sept 2022 to 22 Feb 2026



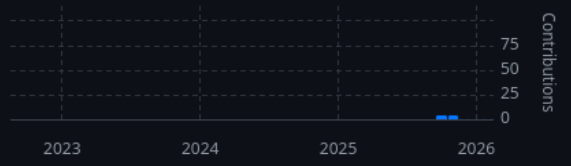
#1 itzmeanjan

523 commits 49,087 ++ 39,046 --



#2 dependabot[bot]

10 commits 10 ++ 10 --



#3 david-cermak

1 commit 2 ++ 1 --

#4 reneleonhardt

1 commit 12 ++ 2 --

```
ml-kem on  master [?] via △ v3.31.6
❯ cloc include/
      14 text files.
      14 unique files.
       0 files ignored.

github.com/AlDanial/cloc v 2.04  T=0.01 s (1105.9 files/s, 144481.4 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C/C++ Header                    14            311            258           1260
-------------------------------------------------------------------------------
SUM:                            14            311            258           1260
-------------------------------------------------------------------------------
```

ml-dsa Public

Unpin | Unwatch 4 | Fork 8 | Star 58

- Pulse
- Contributors
- Community
- Community standards
- Traffic
- Commits
- Code frequency
- Dependency graph
- Network
- Forks
- Actions usage metrics
- Actions performance metrics

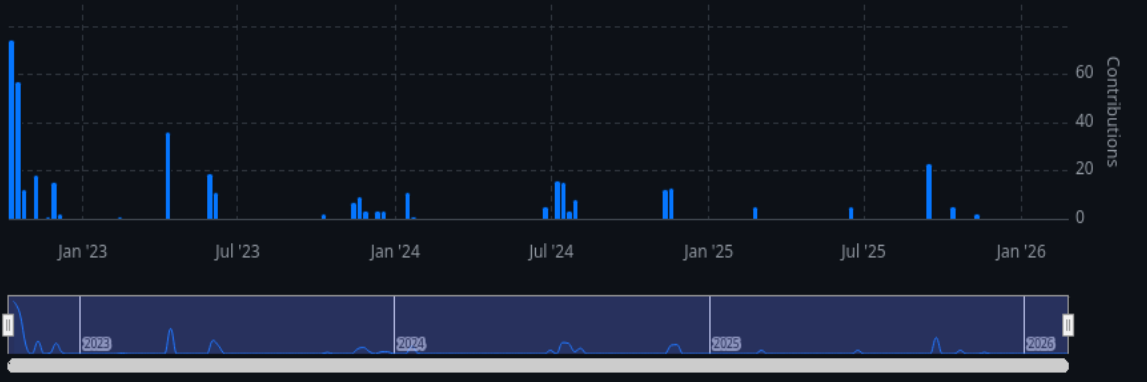
Contributors

Contributions per week to master, excluding merge commits

Period: All | Contributions: Commits

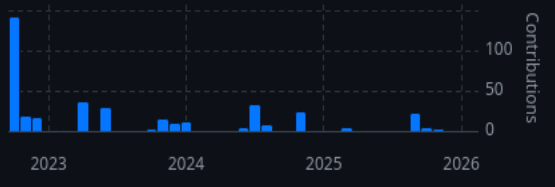
Commits over time

Weekly from 16 Oct 2022 to 22 Feb 2026



itzmeanjan #1

392 commits 40,746 ++ 28,275 --



ElMostafaIdrassi #2

5 commits 195 ++ 195 --



```
ml-dsa on  even-stricter-testing [?] via △ v3.31.6
❯ cloc include/
      15 text files.
      15 unique files.
       0 files ignored.

github.com/AlDanial/cloc v 2.04  T=0.02 s (838.2 files/s, 161553.7 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C/C++ Header                    15            441            446           2004
-------------------------------------------------------------------------------
SUM:                            15            441            446           2004
-------------------------------------------------------------------------------
```

So, what are `ml-kem` and `ml-dsa` C++ lib.?

- Two C++ libraries, implementing FIPS 203, 204
- Zero third-party dependency
- Header-only library
- Fully compile-time evaluable
- NIST ACVP Compliant
- Constant-time ops.

```
ml-kem on  master [?] via △ v3.31.6
❯ tree --du -h --dirsfirst include/
[ 94K]  include/
└── [ 90K]  ml_kem
    ├── [ 78K]  internals
    │   ├── [8.9K]  math
    │   │   └── [4.9K]  field.hpp
    │   ├── [ 39K]  poly
    │   │   ├── [2.6K]  compression.hpp
    │   │   ├── [6.5K]  ntt.hpp
    │   │   ├── [6.1K]  poly_vec.hpp
    │   │   ├── [5.9K]  sampling.hpp
    │   │   └── [ 14K]  serialize.hpp
    │   ├── [ 11K]  utility
    │   │   ├── [ 656]  force_inline.hpp
    │   │   ├── [3.6K]  params.hpp
    │   │   └── [3.1K]  utils.hpp
    │   ├── [7.4K]  k_pke.hpp
    │   └── [7.2K]  ml_kem.hpp
    ├── [2.6K]  ml_kem_1024.hpp
    ├── [2.6K]  ml_kem_512.hpp
    └── [2.6K]  ml_kem_768.hpp

  94K used in 6 directories, 14 files
```

ML-KEM: Supports all three security levels.

```
ml-dsa on  even-stricter-testing [?] via △ v3.31.6
❯ tree --du -h --dirsfirst include/
[137K]  include/
└── [133K]  ml_dsa
    ├── [109K]  internals
    │   ├── [ 16K]  math
    │   │   ├── [7.3K]  field.hpp
    │   │   └── [4.4K]  reduction.hpp
    │   ├── [ 49K]  poly
    │   │   ├── [ 18K]  bit_packing.hpp
    │   │   ├── [4.4K]  ntt.hpp
    │   │   ├── [5.2K]  poly.hpp
    │   │   ├── [ 10K]  polyvec.hpp
    │   │   └── [7.9K]  sampling.hpp
    │   ├── [ 16K]  utility
    │   │   ├── [ 551]  force_inline.hpp
    │   │   ├── [4.9K]  params.hpp
    │   │   ├── [3.7K]  prehash.hpp
    │   │   └── [2.6K]  utils.hpp
    │   └── [ 24K]  ml_dsa.hpp
    ├── [6.7K]  ml_dsa_44.hpp
    ├── [6.7K]  ml_dsa_65.hpp
    └── [6.7K]  ml_dsa_87.hpp

 137K used in 6 directories, 15 files
```

ML-DSA: Supports all three security levels, including pre-hashing mode, with only SHA3 hash fn.

Why C++ though? 🤔

It’s not just C++, it’s modern C++.

What is modern C++?

- Use `std::span`, instead of (raw pointer + size) pair
- Choose `constexpr` over MACROs
- `requires` expression for compile-time param validation
- Do what all can be done in compile-time

Do not forget, `template` meta-programming.
It has been there since C++98.

std::span?

```
39
40  // Computes a new ML-KEM-768 keypair, given seed `d` and `z`.
41  constexpr void
42  keygen(std::span<const uint8_t, SEED_D_BYTE_LEN> d,
43         std::span<const uint8_t, SEED_Z_BYTE_LEN> z,
44         std::span<uint8_t, PKEY_BYTE_LEN> pubkey,
45         std::span<uint8_t, SKEY_BYTE_LEN> seckey)
46  {
47    ml_kem::keygen<k, eta1>(d, z, pubkey, seckey);
48  }
49
50  // Given seed `m` and a ML-KEM-768 public key, this routine computes a ML-KEM-768 cipher text and a fixed size shared secret.
51  // If, input ML-KEM-768 public key is malformed, encapsulation will fail, returning false.
52  [[nodiscard("If public key is malformed, encapsulation fails")]] constexpr bool
53  encapsulate(std::span<const uint8_t, SEED_M_BYTE_LEN> m,
54              std::span<const uint8_t, PKEY_BYTE_LEN> pubkey,
55              std::span<uint8_t, CIPHER_TEXT_BYTE_LEN> cipher,
56              std::span<uint8_t, SHARED_SECRET_BYTE_LEN> shared_secret)
57  {
58    return ml_kem::encapsulate<k, eta1, eta2, du, dv>(m, pubkey, cipher, shared_secret);
59  }
60
61  // Given a ML-KEM-768 secret key and a cipher text, this routine computes a fixed size shared secret.
62  constexpr void
63  decapsulate(std::span<const uint8_t, SKEY_BYTE_LEN> seckey, std::span<const uint8_t, CIPHER_TEXT_BYTE_LEN> cipher, std::span<uint8_t, SHARED_SECRET_BYTE_LEN> shared_secret)
64  {
65    ml_kem::decapsulate<k, eta1, eta2, du, dv>(seckey, cipher, shared_secret);
66  }
67
```

Clean interface.
Statically sized.

Kyber-768 with raw pointers

```
25
26  // Computes a new Kyber768 KEM keypair s.t. public key is 1184 -bytes and secret
27  // key is 2400 -bytes, given 32 -bytes seed d ( used in CPA-PKE ) and 32 -bytes
28  // seed z ( used in CCA-KEM ).
29  inline void
30  keygen(const uint8_t* const __restrict d,
31         const uint8_t* const __restrict z,
32         uint8_t* const __restrict pubkey,
33         uint8_t* const __restrict seckey)
34  {
35    kem::keygen<k, η1>(d, z, pubkey, seckey);
36  }
37
38  // Given 32 -bytes seed m ( which is used during encapsulation ) and a Kyber768
39  // KEM public key ( of 1184 -bytes ), this routine computes a SHAKE256 XOF
40  // backed KDF (key derivation function) and 1088 -bytes of cipher text, which
41  // can only be decrypted by corresponding Kyber768 KEM secret key, for arriving
42  // at same SHAKE256 XOF backed KDF.
43  //
44  // Returned KDF can be used for deriving shared key of arbitrary bytes length.
45  inline shake256::shake256
46  encapsulate(const uint8_t* const __restrict m,
47              const uint8_t* const __restrict pubkey,
48              uint8_t* const __restrict cipher)
49  {
50    return kem::encapsulate<k, η1, η2, du, dv>(m, pubkey, cipher);
51  }
52
53  // Given a Kyber768 KEM secret key ( of 2400 -bytes ) and a cipher text of 1088
54  // -bytes, which holds encrypted ( using corresponding Kyber768 KEM public key )
55  // 32 -bytes seed, this routine computes a SHAKE256 XOF backed KDF (key
56  // derivation function).
57  //
58  // Returned KDF can be used for deriving shared key of arbitrary bytes length.
59  inline shake256::shake256
60  decapsulate(const uint8_t* const __restrict seckey,
61              const uint8_t* const __restrict cipher)
62  {
63    return kem::decapsulate<k, η1, η2, du, dv>(seckey, cipher);
64  }
```

`const` keyword appearing twice.
`__restrict` compiler extension.

Is clean interface all you get with `std::span`?

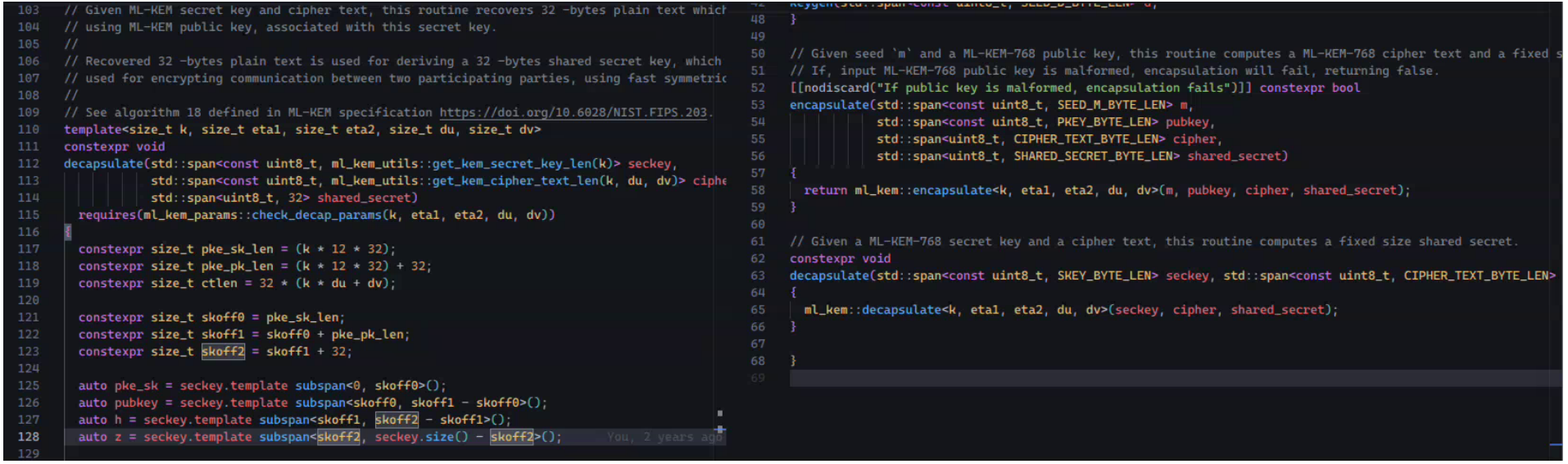
```
19  std::array<uint8_t, ml_kem_512::SEED_M_BYTE_LEN> m{};
20  std::array<uint8_t, ml_kem_512::CIPHER_TEXT_BYTE_LEN> cipher{};
21  std::array<uint8_t, ml_kem_512::SHARED_SECRET_BYTE_LEN> sender_key{};
22
23  csprng.generate(m);
24  assert(ml_kem_512::encapsulate(m, pkey, cipher, sender_key)); // Key Encapsulation might fail, if input public key is malformed
25
```

With `std::span`, no more pointer arithmetic

```
116  {
117    constexpr size_t pke_sk_len = (k * 12 * 32);
118    constexpr size_t pke_pk_len = (k * 12 * 32) + 32;
119    constexpr size_t ctlen = 32 * (k * du + dv);
120
121    constexpr size_t skoff0 = pke_sk_len;
122    constexpr size_t skoff1 = skoff0 + pke_pk_len;
123    constexpr size_t skoff2 = skoff1 + 32;
124
125    auto pke_sk = seckey.template subspan<0, skoff0>();      You, 2 years ago • refactor API surface of Kyber KEM by starting t…
126    auto pubkey = seckey.template subspan<skoff0, skoff1 - skoff0>();
127    auto h = seckey.template subspan<skoff1, skoff2 - skoff1>();
128    auto z = seckey.template subspan<skoff2, seckey.size() - skoff2>();
129
```

Fully compile-time resolved, if subspan is miscalculated.

Compile-time bounds check with `std::span`



```cpp
// Given ML-KEM secret key and cipher text, this routine recovers 32 -bytes plain text which
// using ML-KEM public key, associated with this secret key.
//
// Recovered 32 -bytes plain text is used for deriving a 32 -bytes shared secret key, which
// used for encrypting communication between two participating parties, using fast symmetric
//
// See algorithm 18 defined in ML-KEM specification https://doi.org/10.6028/NIST.FIPS.203.
template<size_t k, size_t eta1, size_t eta2, size_t du, size_t dv>
constexpr void
decapsulate(std::span<const uint8_t, ml_kem_utils::get_kem_secret_key_len(k)> seckey,
            std::span<const uint8_t, ml_kem_utils::get_kem_cipher_text_len(k, du, dv)> ciphe
            std::span<uint8_t, 32> shared_secret)
  requires(ml_kem_params::check_decap_params(k, eta1, eta2, du, dv))
{
  constexpr size_t pke_sk_len = (k * 12 * 32);
  constexpr size_t pke_pk_len = (k * 12 * 32) + 32;
  constexpr size_t ctlen = 32 * (k * du + dv);

  constexpr size_t skoff0 = pke_sk_len;
  constexpr size_t skoff1 = skoff0 + pke_pk_len;
  constexpr size_t skoff2 = skoff1 + 32;

  auto pke_sk = seckey.template subspan<0, skoff0>();
  auto pubkey = seckey.template subspan<skoff0, skoff1 - skoff0>();
  auto h = seckey.template subspan<skoff1, skoff2 - skoff1>();
  auto z = seckey.template subspan<skoff2, seckey.size() - skoff2>();      You, 2 years ago
```

```cpp
}

// Given seed `m` and a ML-KEM-768 public key, this routine computes a ML-KEM-768 cipher text and a fixed s
// If, input ML-KEM-768 public key is malformed, encapsulation will fail, returning false.
[[nodiscard("If public key is malformed, encapsulation fails")]] constexpr bool
encapsulate(std::span<const uint8_t, SEED_M_BYTE_LEN> m,
            std::span<const uint8_t, PKEY_BYTE_LEN> pubkey,
            std::span<uint8_t, CIPHER_TEXT_BYTE_LEN> cipher,
            std::span<uint8_t, SHARED_SECRET_BYTE_LEN> shared_secret)
{
  return ml_kem::encapsulate<k, eta1, eta2, du, dv>(m, pubkey, cipher, shared_secret);
}

// Given a ML-KEM-768 secret key and a cipher text, this routine computes a fixed size shared secret.
constexpr void
decapsulate(std::span<const uint8_t, SKEY_BYTE_LEN> seckey, std::span<const uint8_t, CIPHER_TEXT_BYTE_LEN>
{
  ml_kem::decapsulate<k, eta1, eta2, du, dv>(seckey, cipher, shared_secret);
}

}
```

Is `std::span` a silver bullet then?

```
20  std::array<uint8_t, ml_kem_512::SEED_M_BYTE_LEN> m{};
21  std::array<uint8_t, ml_kem_512::CIPHER_TEXT_BYTE_LEN> cipher{};
22  std::array<uint8_t, ml_kem_512::SHARED_SECRET_BYTE_LEN> sender_key{};
23
24  // A crafted span, which is designed to look like 32 bytes wide, though not
25  auto* sender_key_ptr = sender_key.data();
26  auto sender_key_span = std::span<uint8_t, ml_kem_512::SHARED_SECRET_BYTE_LEN>(sender_key_ptr, ml_kem_512::SHARED_SECRET_BYTE_LEN);
27
28  csprng.generate(m);
29  assert(ml_kem_512::encapsulate(m, pkey, cipher, sender_key_span)); // Key Encapsulation might fail, if input public key is malformed
```

Just harden the binary, if needed

Fails to catch it

```
ml-kem on  master [?] via △ v3.31.6
❯ c++ -std=c++20 -O3 -Iinclude -Isha3/include -Isubtle/include -IRandomShake/include main.cpp  && ./a.out

ml-kem on  master [?] via △ v3.31.6 took 4s
❯ c++ -std=c++20 -O3 -Iinclude -Isha3/include -Isubtle/include -IRandomShake/include -D_GLIBCXX_ASSERTIONS main.cpp  && ./a.out
/usr/include/c++/15/span:83: constexpr std::__detail::__extent_storage<_Extent>::__extent_storage(std::size_t) [with long unsigned int _Extent = 32; std::size_t = long unsigned int]: Assertion '__n == _Extent' failed.
Aborted (core dumped)
```

Catches it

For GCC : `-D_GLIBCXX_ASSERTIONS`

For Clang: `-D_LIBCPP_HARDENING_MODE=_LIBCPP_HARDENING_MODE_EXTENSIVE`

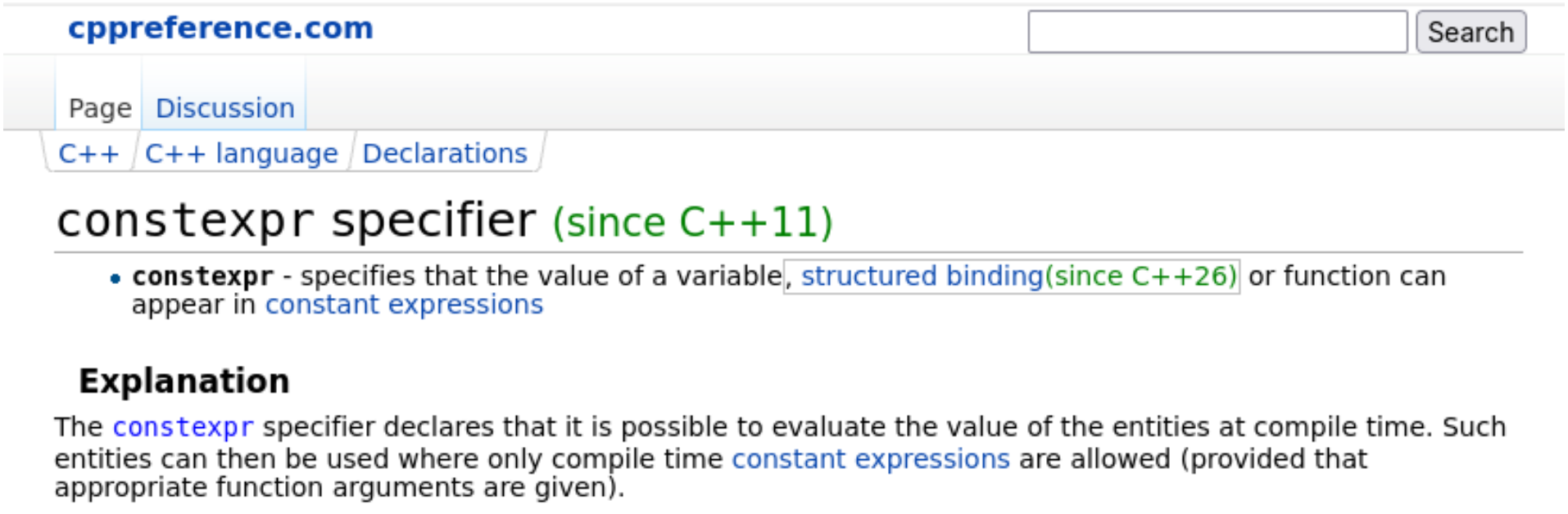
What is the deal with compile-time eval.?

- Compute output of any ML-KEM or ML-DSA fn. at compile-time itself.
- Input values must be known at compile-time.
- Compiler should have sufficient `constexpr-steps` allowance.

Compile-time ML-KEM-512 in action

```
5   // If this file compiles, the static_assert has already proven correctness at compile-time.
    You, 8 seconds ago | 1 author (You)
6   namespace {
7
8   // KAT vector no. 1: seeds and expected shared secrets.
9   constexpr auto SEED_D = from_hex<32>("7c9935a0b07694aa0c6d10e4db6b1add2fd81a25ccb148032dcd739936737f2d");
10  constexpr auto SEED_Z = from_hex<32>("b505d7cfad1b497499323c8686325e4792f267aafa3f87ca60d01cb54f29202a");
11  constexpr auto SEED_M = from_hex<32>("eb4a7c66ef4eba2ddb38c88d8bc706b1d639002198172a7b1942eca8f6c001ba");
12  constexpr auto EXPECTED_SS = from_hex<32>("b4c8e3c4115f9511f2fddb288c4b78c5cd7c89d2d4d321f46b4edc54ddf0eb36");
13
14  // Compile-time evaluation of ML-KEM-512 keygen, encapsulation, and decapsulation.
15  constexpr bool
16  eval_ml_kem_512()
17  {
18    std::array<uint8_t, ml_kem_512::PKEY_BYTE_LEN> pubkey{};
19    std::array<uint8_t, ml_kem_512::SKEY_BYTE_LEN> seckey{};
20    std::array<uint8_t, ml_kem_512::CIPHER_TEXT_BYTE_LEN> cipher{};
21    std::array<uint8_t, ml_kem_512::SHARED_SECRET_BYTE_LEN> sender_key{};
22    std::array<uint8_t, ml_kem_512::SHARED_SECRET_BYTE_LEN> receiver_key{};
23
24    ml_kem_512::keygen(SEED_D, SEED_Z, pubkey, seckey);
25    const auto encaps_ok = ml_kem_512::encapsulate(SEED_M, pubkey, cipher, sender_key);
26    ml_kem_512::decapsulate(seckey, cipher, receiver_key);
27
28    return encaps_ok && (sender_key == receiver_key) && (sender_key == EXPECTED_SS);
29  }
30
31  } // anonymous namespace
32
33  // The static_asserts below are the real tests. They fire at compile-time.
34  // If this translation unit compiles, all three ML-KEM variants are proven correct at program compile-time itself.
35  // The GTest bodies exist only so that ctest reports them as passed.
36
37  TEST(ML_KEM, CompileTimeEval_ML_KEM_512)
38  {
39    static_assert(eval_ml_kem_512(), "ML-KEM-512 keygen/encaps/decaps must be correct at compile-time");
40    SUCCEED();
41  }
```

constexpr?



cppreference.com

Search

Page Discussion

C++ C++ language Declarations

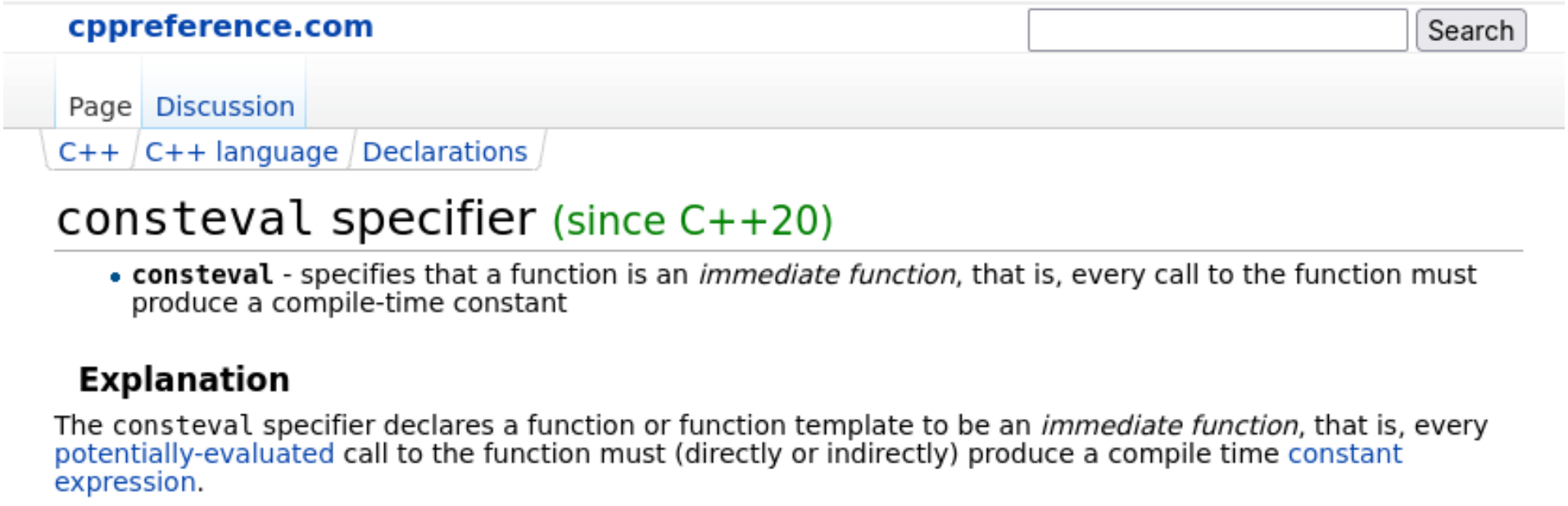
constexpr specifier (since C++11)

- **constexpr** - specifies that the value of a variable, structured binding(since C++26) or function can appear in constant expressions

Explanation

The constexpr specifier declares that it is possible to evaluate the value of the entities at compile time. Such entities can then be used where only compile time constant expressions are allowed (provided that appropriate function arguments are given).

consteval?



cppreference.com

Search

Page Discussion

C++ C++ language Declarations

consteval specifier (since C++20)

- **consteval** - specifies that a function is an *immediate function*, that is, every call to the function must produce a compile-time constant

Explanation

The consteval specifier declares a function or function template to be an *immediate function*, that is, every potentially-evaluated call to the function must (directly or indirectly) produce a compile time constant expression.

Parameter validation fn. is `consteval`

```
59  // Compile-time check to ensure that K-PKE encryption routine is invoked with proper parameter set.
60  //
61  // See table 2 of ML-KEM specification https://doi.org/10.6028/NIST.FIPS.203.
62  consteval bool
63  check_encrypt_params(const size_t k, const size_t eta1, const size_t eta2, const size_t du, const size_t dv)
64  {
65    const bool flg0 = (k == 2) && (eta1 == 3) && (eta2 == 2) && (du == 10) && (dv == 4);
66    const bool flg1 = (k == 3) && (eta1 == 2) && (eta2 == 2) && (du == 10) && (dv == 4);
67    const bool flg2 = (k == 4) && (eta1 == 2) && (eta2 == 2) && (du == 11) && (dv == 5);
68
69    return flg0 || flg1 || flg2;
70  }
71
72  // Compile-time check to ensure that ML-KEM encapsulation routine is invoked with proper parameter set.
73  //
74  // See table 2 of ML-KEM specification https://doi.org/10.6028/NIST.FIPS.203.
75  consteval bool
76  check_encap_params(const size_t k, const size_t eta1, const size_t eta2, const size_t du, const size_t dv)
77  {
78    return check_encrypt_params(k, eta1, eta2, du, dv);
79  }       You, now • Uncommitted changes
80
```

We want `consteval` for param. validation

```
56   // See algorithm 17 defined in ML-KEM specification https://doi.org/10.6028/NIST.FIPS.203.
57   template<size_t k, size_t eta1, size_t eta2, size_t du, size_t dv>
58   [[nodiscard("Use result, it might fail because of malformed input public key")]] constexpr bool
59 > encapsulate(std::span<const uint8_t, 32> m, …
62                 std::span<uint8_t, 32> shared_secret)
63     requires(ml_kem_params::check_encap_params(k, eta1, eta2, du, dv))      You, 20 months ago • R
64 > {…
101  }
```

But we want `constexpr` for crypto. impl. itself

```
56   // See algorithm 17 defined in ML-KEM specification https://doi.org/10.6028/NIST.FIPS.203.
57   template<size_t k, size_t eta1, size_t eta2, size_t du, size_t dv>
58   [[nodiscard("Use result, it might fail because of malformed input public key")]] constexpr bool
59   encapsulate(std::span<const uint8_t, 32> m,
60               std::span<const uint8_t, ml_kem_utils::get_kem_public_key_len(k)> pubkey,
61               std::span<uint8_t, ml_kem_utils::get_kem_cipher_text_len(k, du, dv)> cipher,
62               std::span<uint8_t, 32> shared_secret)
63     requires(ml_kem_params::check_encap_params(k, eta1, eta2, du, dv))
64 > {…
101  }
```

For `encaps.` to be `constexpr`,
Every computation it does need to be `constexpr` too.

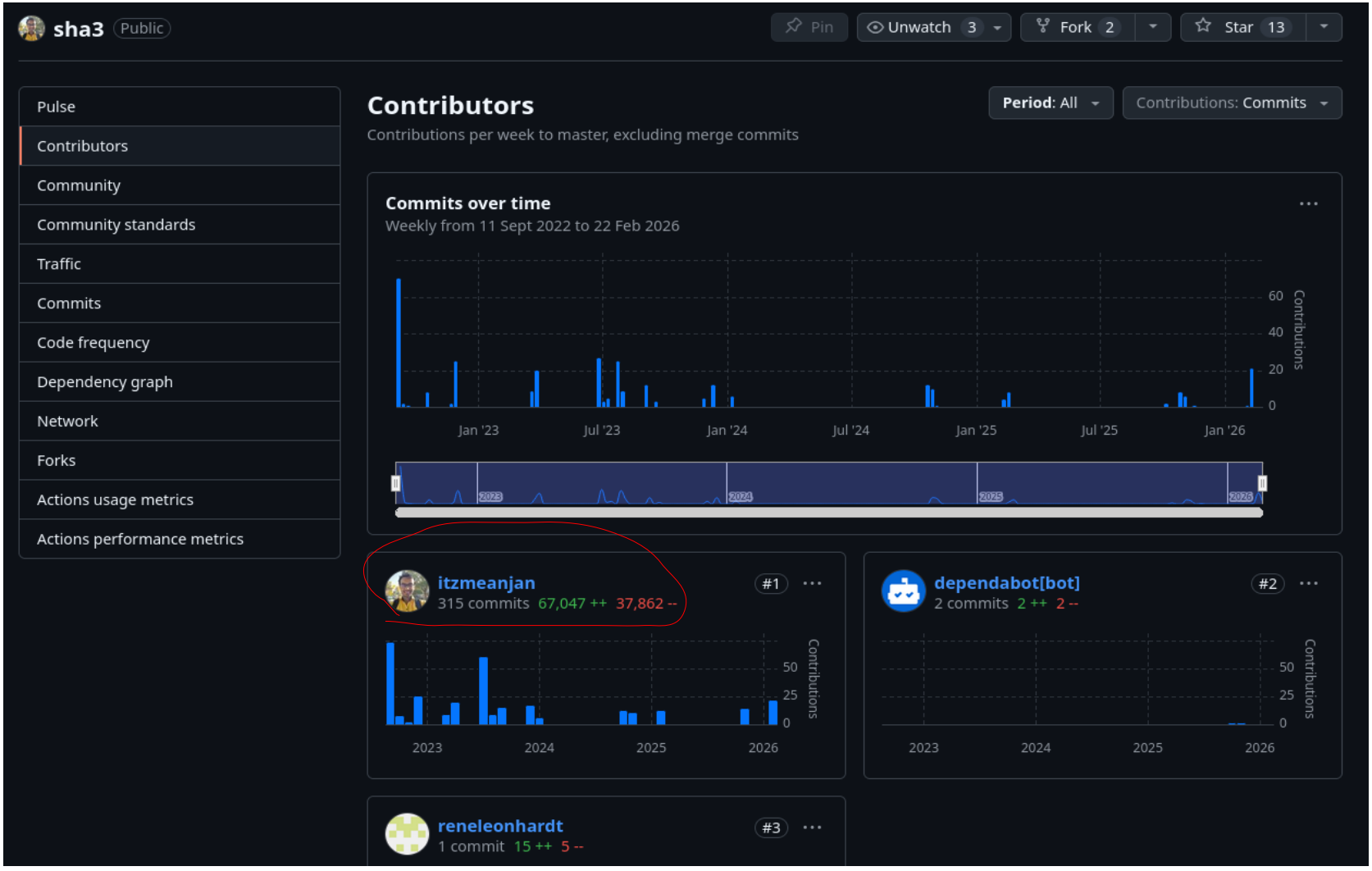
Is hashing `constexpr` too?

Yes 🤩

Introducing `sha3`. Yet another implementation of **NIST FIPS 202**.
It is a modern C++, header-only, fully `constexpr` library impl. of SHA3 + TurboSHAKE.

Check out **RFC 9861**.
It's ~2x faster than SHAKE.
And much faster than SHA3.

Quick detour, sha3



Compile-time evaluation and embedding of SHA3-256 message digest

```
51
52  constexpr std::string_view MSG = "keccak permutation rocks!";
53  constexpr auto MSG_bytes = string_to_bytes<MSG.size()>(MSG);
54  constexpr auto MD = sha3_256::sha3_256_t::hash(MSG_bytes);
55
56  int
57  main()
58  {
59    constexpr auto expected_md = from_hex<sha3_256::DIGEST_LEN>("4edc60a9ffe739ce44252716483a529e8a859a5a75cbf69d494037e914bac16b");
60    static_assert(MD == expected_md, "Must compute SHA3-256 message digest in program compile-time!");
61
62    return EXIT_SUCCESS;
63  }
64
```

See static assertion!

We take `constexpr` to next level, in `sha3` lib.

```cpp
/**
 * Computes 64 -bit round constant (at compile-time), which is XOR-ed into the very first lane ( = lane(0, 0) ) of Keccak-f[1600] permutation state.
 *
 * Taken from https://github.com/itzmeanjan/elephant/blob/2a21c7e/include/keccak.hpp#L61-L74.
 */
static consteval uint64_t
compute_rc(const size_t r_idx)
{
  uint64_t tmp = 0;

  for (size_t j = 0; j < (L + 1); j++) {
    const size_t boff = (1 << j) - 1;
    tmp |= static_cast<uint64_t>(rc(j + (7 * r_idx))) << boff;
  }

  return tmp;
}

// Compile-time evaluate Keccak-f[1600] round constants.
static consteval std::array<uint64_t, MAX_NUM_ROUNDS>
compute_rcs()
{
  std::array<uint64_t, MAX_NUM_ROUNDS> res{};

  for (size_t i = 0; i < MAX_NUM_ROUNDS; i++) {
    res[i] = compute_rc(i);
  }

  return res;
}

// Round constants to be XORed with lane (0, 0) of Keccak-f[1600] permutation state. See section 3.2.5 of https://dx.doi.org/10.6028/NIST.FIPS.202.
static constexpr std::array<uint64_t, MAX_NUM_ROUNDS> RC = compute_rcs();
```

Why hardcode keccak round-constants, when we can compute them at program compile time?

But how does `sha3` compare to `xkcp`?

```
benchmark/tools on  main via 🐍 v3.13.7
❯ ./compare.py filters ~/Documents/my_work/sha3/turboshake.json xkcp_turboshake128 portable_turboshake128
Comparing xkcp_turboshake128 to portable_turboshake128 (from /home/anjan/Documents/my_work/sha3/turboshake.json)
Benchmark                                                                                  Time             CPU      Time Old      Time New       CPU Old       CPU New
-----------------------------------------------------------------------------------------------------------------------------------------------------------------
[xkcp_turboshake128 vs. portable_turboshake128]/32/64_mean                              -0.0491         -0.0491           113           107           113           107
[xkcp_turboshake128 vs. portable_turboshake128]/32/64_median                            -0.0471         -0.0470           113           107           113           107
[xkcp_turboshake128 vs. portable_turboshake128]/32/64_stddev                            +0.0784         +0.0603             1             1             1             1
[xkcp_turboshake128 vs. portable_turboshake128]/32/64_cv                                +0.1341         +0.1150             0             0             0             0
[xkcp_turboshake128 vs. portable_turboshake128]/32/64_min                               -0.0521         -0.0518           111           105           111           105
[xkcp_turboshake128 vs. portable_turboshake128]/32/64_max                               -0.0494         -0.0495           115           109           115           109
[xkcp_turboshake128 vs. portable_turboshake128]/1024/64_mean                            +0.1052         +0.1051           694           767           694           767
[xkcp_turboshake128 vs. portable_turboshake128]/1024/64_median                          +0.1072         +0.1072           694           768           693           768
[xkcp_turboshake128 vs. portable_turboshake128]/1024/64_stddev                          +0.1424         +0.1533             4             5             4             5
[xkcp_turboshake128 vs. portable_turboshake128]/1024/64_cv                              +0.0337         +0.0435             0             0             0             0
[xkcp_turboshake128 vs. portable_turboshake128]/1024/64_min                             +0.1050         +0.1046           687           759           687           759
[xkcp_turboshake128 vs. portable_turboshake128]/1024/64_max                             +0.1034         +0.1033           701           774           701           773
[xkcp_turboshake128 vs. portable_turboshake128]/32768/64_mean                           +0.1407         +0.1408         18745         21382         18742         21380
[xkcp_turboshake128 vs. portable_turboshake128]/32768/64_median                         +0.1405         +0.1406         18712         21342         18711         21341
[xkcp_turboshake128 vs. portable_turboshake128]/32768/64_stddev                         +0.2578         +0.2579           147           185           147           185
[xkcp_turboshake128 vs. portable_turboshake128]/32768/64_cv                             +0.1026         +0.1027             0             0             0             0
[xkcp_turboshake128 vs. portable_turboshake128]/32768/64_min                            +0.1424         +0.1425         18549         21191         18548         21190
[xkcp_turboshake128 vs. portable_turboshake128]/32768/64_max                            +0.1445         +0.1447         19013         21760         19010         21760
[xkcp_turboshake128 vs. portable_turboshake128]/1048576/64_mean                         +0.1640         +0.1639        595362        692988        595272        692860
[xkcp_turboshake128 vs. portable_turboshake128]/1048576/64_median                       +0.1674         +0.1674        594830        694396        594781        694339
[xkcp_turboshake128 vs. portable_turboshake128]/1048576/64_stddev                       -0.4135         -0.3981          7035          4126          7029          4231
[xkcp_turboshake128 vs. portable_turboshake128]/1048576/64_cv                           -0.4962         -0.4829             0             0             0             0
[xkcp_turboshake128 vs. portable_turboshake128]/1048576/64_min                          +0.1690         +0.1683        587106        686356        587057        685856
[xkcp_turboshake128 vs. portable_turboshake128]/1048576/64_max                          +0.1413         +0.1414        612254        698771        612166        698713
[xkcp_turboshake128 vs. portable_turboshake128]/33554432/64_mean                        +0.1399         +0.1399      19699389      22455442      19697166      22452169
[xkcp_turboshake128 vs. portable_turboshake128]/33554432/64_median                      +0.1366         +0.1364      19677348      22364807      19675309      22358679
[xkcp_turboshake128 vs. portable_turboshake128]/33554432/64_stddev                      +0.1169         +0.1186        196616        219601        196216        219494
[xkcp_turboshake128 vs. portable_turboshake128]/33554432/64_cv                          -0.0202         -0.0186             0             0             0             0
[xkcp_turboshake128 vs. portable_turboshake128]/33554432/64_min                         +0.1458         +0.1457      19426741      22258610      19425325      22255352
[xkcp_turboshake128 vs. portable_turboshake128]/33554432/64_max                         +0.1512         +0.1512      19972864      22992716      19970132      22989337
[xkcp_turboshake128 vs. portable_turboshake128]/1073741824/64_mean                      +0.1545         +0.1545     625210638     721781206     625124456     721714881
[xkcp_turboshake128 vs. portable_turboshake128]/1073741824/64_median                    +0.1559         +0.1560     624065785     721373519     623997399     721315537
[xkcp_turboshake128 vs. portable_turboshake128]/1073741824/64_stddev                    -0.6865         -0.6862       8189665       2567286       8196793       2572138
[xkcp_turboshake128 vs. portable_turboshake128]/1073741824/64_cv                        -0.7285         -0.7282             0             0             0             0
[xkcp_turboshake128 vs. portable_turboshake128]/1073741824/64_min                       +0.1732         +0.1732     612086397     718115792     612006563     718028910
[xkcp_turboshake128 vs. portable_turboshake128]/1073741824/64_max                       +0.1326         +0.1326     641148317     726144521     641108415     726097191

benchmark/tools on  main via 🐍 v3.13.7
❯
```

- Takes ~(10-15)% more time than XKCP 🥲
- But it is easier to use
- It is minimal
- It is plain C++
- No platform specific optimization, yet 😉
- Couple of loop unrolls
- And compile with `-O3 -march=native`

Still interested in `sha3`?



Back to PQC

Benefit of having full compile-time eval. Zq arithmetic type

```
You, last week | 1 author (You)
namespace ml_kem_ntt {

inline constexpr size_t LOG2N = 8;
inline constexpr size_t N = 1 << LOG2N;

// First primitive 256 -th root of unity modulo q | q = 3329
//
// Meaning, 17 ** 256 == 1 mod q
inline constexpr auto ZETA = ml_kem_field::zq_t(17);
static_assert((ZETA ^ N) == ml_kem_field::zq_t::one(), "ZETA must be 256th root of unity modulo Q");

// Multiplicative inverse of N/ 2 over Z_q | q = 3329 and N = 256
//
// Meaning (N/ 2) * INV_N = 1 mod q
inline constexpr auto INV_N = ml_kem_field::zq_t(N / 2).inv();
```

Instead of hardcoding a table of 128 powers of `zeta` values, just compile-time compute them

```
41
42  // Compile-time computed constants ( powers of ζ ), used for polynomial evaluation i.e. computation of NTT form.
43  inline constexpr std::array<ml_kem_field::zq_t, N / 2> NTT_ZETA_EXP = []() -> auto {      You, last week • Get ri
44  ✦std::array<ml_kem_field::zq_t, N / 2> res{};
45
46    for (size_t i = 0; i < res.size(); i++) {
47      res[i] = ZETA ^ bit_rev<LOG2N - 1>(i);
48    }
49
50    return res;
51  }();
52
```

`constexpr` any good for other than just compile-time eval.?

```
32  struct zq_t
33  {
34  private:
35    // Underlying value held in this type.
36    //
37    // Note, v is always kept in its canonical form i.e. v ∈ [0, Q).
38    uint32_t v = 0U;
39
40    // Given a 32 -bit unsigned integer `v` such that `v` ∈ [0, 2*Q), r
41    static forceinline constexpr uint32_t reduce_once(const uint32_t v)
42    {
43      const uint32_t t0 = v - Q;
44      const uint32_t t1 = 0U - (t0 >> 31);
45      const uint32_t t2 = Q & t1;
46      const uint32_t t3 = t0 + t2;
47
48      return t3;
49
50      // Undefined Behavior
51      // int a = std::numeric_limits<int>::max();
52      // a += 2;
53
54      // const uint32_t t4 = t3 * static_cast<uint32_t>(a);
55      // return t4;
56    }
```

```
3   #include "ml_kem/internals/utility/force_inline.hpp" // IWYU pragma: keep
4   #include <array>
5   #include <cstddef>
6   #include <span>
7
    You, last week | 1 author (You)
8   namespace ml_kem_ntt {
9
10  inline constexpr size_t LOG2N = 8;
11  inline constexpr size_t N = 1 << LOG2N;
12
13  // First primitive 256 -th root of unity modulo q | q = 3329
14  //
15  // Meaning, 17 ** 256 == 1 mod q
16  inline constexpr auto ZETA = ml_kem_field::zq_t(17);
17  static_assert((ZETA ^ N) == ml_kem_field::zq_t::one(), "ZETA must be 256th r
18
19  // Multiplicative inverse of N/ 2 over Z_q | q = 3329 and N = 256
20  //
21  // Meaning (N/ 2) * INV_N = 1 mod q
22  inline constexpr auto INV_N = ml_kem_field::zq_t(N / 2).inv();      You, 17
23
24  // Given a 64 -bit unsigned integer, this routine extracts specified many co
25  // and reverses their bit order, returning bit reversed `mbw` -bit wide numb
26  //
```

You can’t have undefined behavior (UB) in `constexpr` fn.

No UB? Is not that a feature we all seek?

6. an expression whose evaluation leads to any form of core language undefined or erroneous(since C++26) behavior, except for any potential undefined behavior introduced by standard attributes.

```
constexpr double d1 = 2.0 / 1.0; // OK
constexpr double d2 = 2.0 / 0.0; // Error: not defined
constexpr int n = std::numeric_limits<int>::max() + 1; // Error: overflow
int x, y, z[30];
constexpr auto e1 = &y - &x;         // Error: undefined
constexpr auto e2 = &z[20] - &z[3]; // OK
constexpr std::bitset<2> a;
constexpr bool b = a[2]; // UB, but unspecified if detected
```

Taken from cppreference.com

So, in summary?

“A crypto library where the compiler itself verifies correctness - no undefined behavior, no hidden bugs - and every build proves the implementation matches the spec.”

- Gemini 3.1 Pro

Seriously, tell me

- Minimal impl. of NIST FIPS 203, 204
- Plain C++
- No assembly
- Zero third-party dependency
- No UB
- Reduced supply chain attack vector
- Compile-time self-tested against NIST ACVP KATs

If no UB, how can you scrub secret material?

```
You, last week | 1 author (You)
10  namespace ml_kem_utils {
11
12  // Securely zeroizes a std::array, preventing the compiler from optimizing away the operation.
13  // At compile-time: a plain fill suffices -- there is no real memory to protect.
14  // At runtime: the fill is followed by a compiler barrier (empty asm with "memory" clobber)
15  // that forces the compiler to treat the array's memory as externally observable, preventing
16  // dead store elimination. The barrier generates zero instructions.
17  template<typename T, size_t N>
18  forceinline constexpr void
19  secure_zeroize(std::array<T, N>& arr)
20  {
21      arr.fill(T{});
22      if (!std::is_constant_evaluated()) {
23          asm volatile("" : : "r"(arr.data()) : "memory"); // NOLINT(hicpp-no-assembler)
24      }       You, last week • Add function for zeroizing intermediate secret …
25  }
```

CT = compile-time or constant-time?

```cpp

// Given two byte arrays of equal length, this routine can be used for comparing them in constant-time,
// producing truth value (0xffffffff) in case of equality, otherwise it returns false value (0x00000000).
template<size_t n>
forceinline constexpr uint32_t
ct_memcmp(std::span<const uint8_t, n> bytes0, std::span<const uint8_t, n> bytes1)
{
  uint32_t flag = -1U;
  for (size_t i = 0; i < n; i++) {
    flag &= subtle::ct_eq<uint8_t, uint32_t>(bytes0[i], bytes1[i]);
  }

  return flag;
}

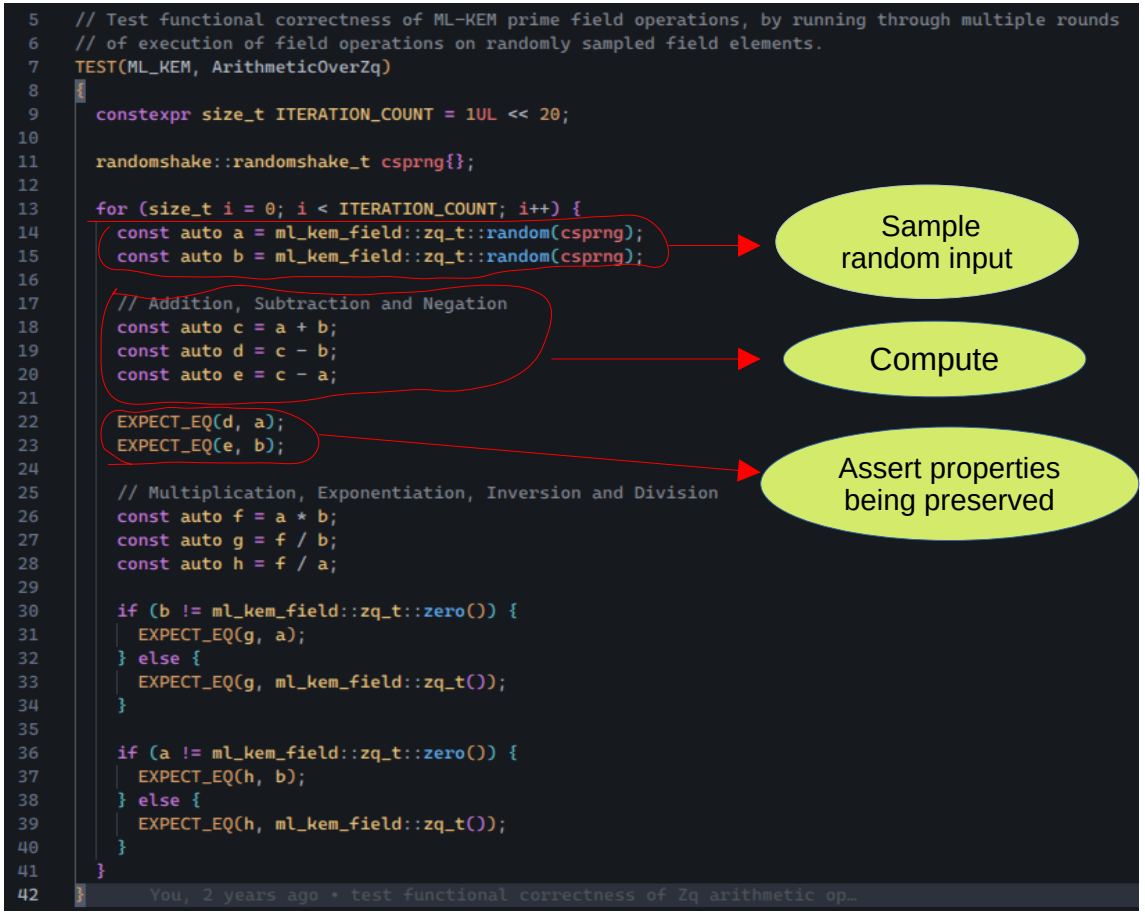
// Given a branch value, taking either 0x00000000 (false value) or 0xffffffff (truth value), this routine can be used for conditionally
// copying bytes from either `source0` byte array (in case branch holds truth value) or `source1` byte array (if branch holds false value)
// to `sink` byte array, all in constant-time.
//
// In simple words, `sink = cond ? source0 ? source1`
template<size_t n>
forceinline constexpr void
ct_cond_memcpy(const uint32_t cond, std::span<uint8_t, n> sink, std::span<const uint8_t, n> source0, std::span<const uint8_t, n> source1)
{
  for (size_t i = 0; i < n; i++) {
    sink[i] = subtle::ct_select(cond, source0[i], source1[i]);
  }
}
```

`ml-kem` and `ml-dsa` ✅ both.

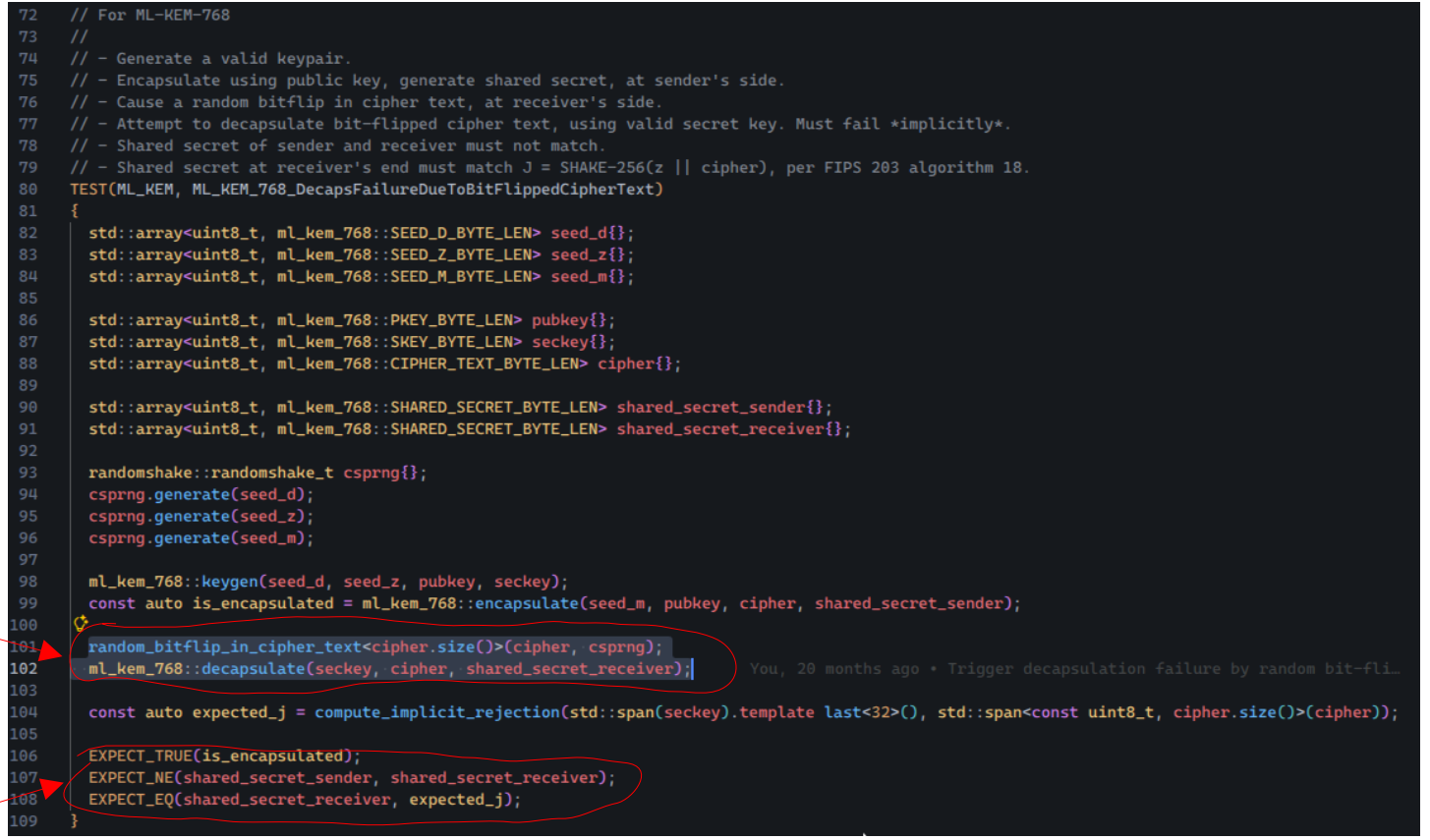
How do you test them?

- No unit testing
- Instead property-based testing
- Introduce pseudo-random mutation
- Assert property breaking or being preserved
- Test against all NIST ACVP KATs
- Run all tests with `(ASAN, UBSAN) x (GCC, Clang) x (Debug, Release, RelWithDebInfo,MinSizeRel) x (macOS, Linux)`
- And fuzz them all! 🚀

What do you mean no unit testing?



What about exercising unhappy path?



```
72  // For ML-KEM-768
73  //
74  // - Generate a valid keypair.
75  // - Encapsulate using public key, generate shared secret, at sender's side.
76  // - Cause a random bitflip in cipher text, at receiver's side.
77  // - Attempt to decapsulate bit-flipped cipher text, using valid secret key. Must fail *implicitly*.
78  // - Shared secret of sender and receiver must not match.
79  // - Shared secret at receiver's end must match J = SHAKE-256(z || cipher), per FIPS 203 algorithm 18.
80  TEST(ML_KEM, ML_KEM_768_DecapsFailureDueToBitFlippedCipherText)
81  {
82    std::array<uint8_t, ml_kem_768::SEED_D_BYTE_LEN> seed_d{};
83    std::array<uint8_t, ml_kem_768::SEED_Z_BYTE_LEN> seed_z{};
84    std::array<uint8_t, ml_kem_768::SEED_M_BYTE_LEN> seed_m{};
85
86    std::array<uint8_t, ml_kem_768::PKEY_BYTE_LEN> pubkey{};
87    std::array<uint8_t, ml_kem_768::SKEY_BYTE_LEN> seckey{};
88    std::array<uint8_t, ml_kem_768::CIPHER_TEXT_BYTE_LEN> cipher{};
89
90    std::array<uint8_t, ml_kem_768::SHARED_SECRET_BYTE_LEN> shared_secret_sender{};
91    std::array<uint8_t, ml_kem_768::SHARED_SECRET_BYTE_LEN> shared_secret_receiver{};
92
93    randomshake::randomshake_t csprng{};
94    csprng.generate(seed_d);
95    csprng.generate(seed_z);
96    csprng.generate(seed_m);
97
98    ml_kem_768::keygen(seed_d, seed_z, pubkey, seckey);
99    const auto is_encapsulated = ml_kem_768::encapsulate(seed_m, pubkey, cipher, shared_secret_sender);
100
101   random_bitflip_in_cipher_text<cipher.size()>(cipher, csprng);
102   ml_kem_768::decapsulate(seckey, cipher, shared_secret_receiver);   You, 20 months ago • Trigger decapsulation failure by random bit-fli…
103
104   const auto expected_j = compute_implicit_rejection(std::span(seckey).template last<32>(), std::span<const uint8_t, cipher.size()>(cipher));
105
106   EXPECT_TRUE(is_encapsulated);
107   EXPECT_NE(shared_secret_sender, shared_secret_receiver);
108   EXPECT_EQ(shared_secret_receiver, expected_j);
109 }
```

Cause a random bit-flip

Assert expected behavior

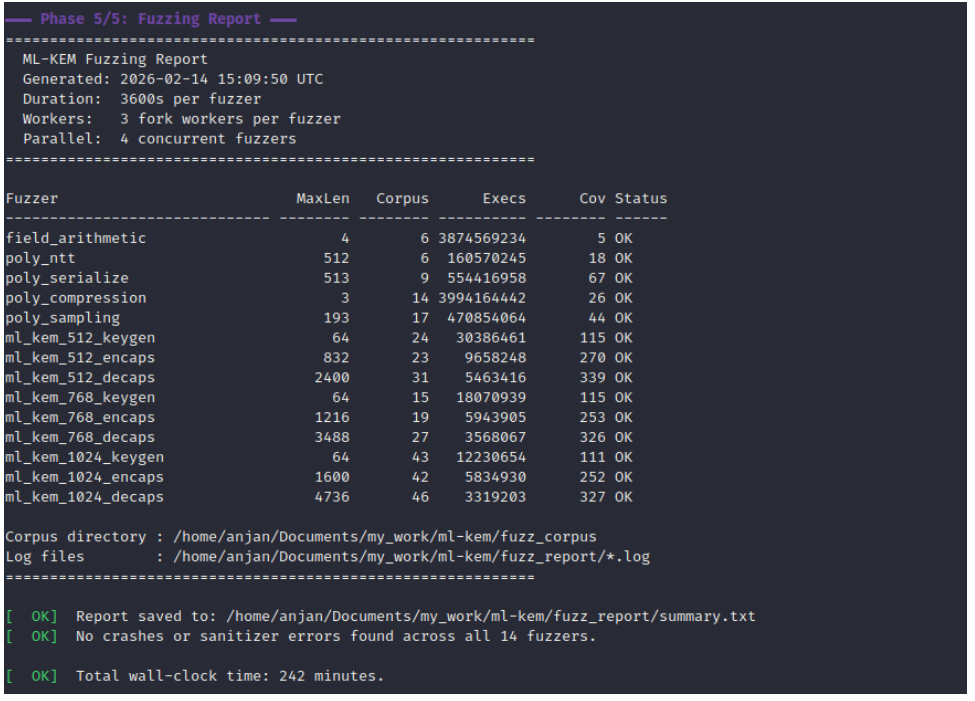
NIST FIPS Compliant: All ACVP KATs ✅

```
❯ ctest --test-dir build --output-on-failure -j -R ACVP
Internal ctest changing into directory: /home/anjan/Documents/my_work/ml-kem/build
Test project /home/anjan/Documents/my_work/ml-kem/build
      Start  2: ML_KEM.ML_KEM_1024_Keygen_ACVP_KnownAnswerTests
      Start  3: ML_KEM.ML_KEM_1024_Encaps_ACVP_KnownAnswerTests
      Start  4: ML_KEM.ML_KEM_1024_Decaps_ACVP_KnownAnswerTests
      Start  5: ML_KEM.ML_KEM_1024_EncapsKeyCheck_ACVP_KnownAnswerTests
      Start  6: ML_KEM.ML_KEM_1024_DecapsKeyCheck_ACVP_KnownAnswerTests
      Start  8: ML_KEM.ML_KEM_512_Keygen_ACVP_KnownAnswerTests
      Start 14: ML_KEM.ML_KEM_768_Keygen_ACVP_KnownAnswerTests
      Start 15: ML_KEM.ML_KEM_768_Encaps_ACVP_KnownAnswerTests
      Start  9: ML_KEM.ML_KEM_512_Encaps_ACVP_KnownAnswerTests
      Start 16: ML_KEM.ML_KEM_768_Decaps_ACVP_KnownAnswerTests
      Start 10: ML_KEM.ML_KEM_512_Decaps_ACVP_KnownAnswerTests
      Start 17: ML_KEM.ML_KEM_768_EncapsKeyCheck_ACVP_KnownAnswerTests
      Start 11: ML_KEM.ML_KEM_512_EncapsKeyCheck_ACVP_KnownAnswerTests
      Start 18: ML_KEM.ML_KEM_768_DecapsKeyCheck_ACVP_KnownAnswerTests
      Start 12: ML_KEM.ML_KEM_512_DecapsKeyCheck_ACVP_KnownAnswerTests
 1/15 Test  #2: ML_KEM.ML_KEM_1024_Keygen_ACVP_KnownAnswerTests ...........   Passed    0.02 sec
 2/15 Test  #3: ML_KEM.ML_KEM_1024_Encaps_ACVP_KnownAnswerTests ...........   Passed    0.02 sec
 3/15 Test  #4: ML_KEM.ML_KEM_1024_Decaps_ACVP_KnownAnswerTests ...........   Passed    0.02 sec
 4/15 Test  #5: ML_KEM.ML_KEM_1024_EncapsKeyCheck_ACVP_KnownAnswerTests ...   Passed    0.01 sec
 5/15 Test  #6: ML_KEM.ML_KEM_1024_DecapsKeyCheck_ACVP_KnownAnswerTests ...   Passed    0.01 sec
 6/15 Test  #8: ML_KEM.ML_KEM_512_Keygen_ACVP_KnownAnswerTests ............   Passed    0.01 sec
 7/15 Test #14: ML_KEM.ML_KEM_768_Keygen_ACVP_KnownAnswerTests ............   Passed    0.01 sec
 8/15 Test #15: ML_KEM.ML_KEM_768_Encaps_ACVP_KnownAnswerTests ............   Passed    0.01 sec
 9/15 Test  #9: ML_KEM.ML_KEM_512_Encaps_ACVP_KnownAnswerTests ............   Passed    0.01 sec
10/15 Test #16: ML_KEM.ML_KEM_768_Decaps_ACVP_KnownAnswerTests ............   Passed    0.01 sec
11/15 Test #10: ML_KEM.ML_KEM_512_Decaps_ACVP_KnownAnswerTests ............   Passed    0.01 sec
12/15 Test #17: ML_KEM.ML_KEM_768_EncapsKeyCheck_ACVP_KnownAnswerTests ....   Passed    0.01 sec
13/15 Test #11: ML_KEM.ML_KEM_512_EncapsKeyCheck_ACVP_KnownAnswerTests ....   Passed    0.00 sec
14/15 Test #18: ML_KEM.ML_KEM_768_DecapsKeyCheck_ACVP_KnownAnswerTests ....   Passed    0.00 sec
15/15 Test #12: ML_KEM.ML_KEM_512_DecapsKeyCheck_ACVP_KnownAnswerTests ....   Passed    0.00 sec

100% tests passed, 0 tests failed out of 15

Total Test time (real) =   0.02 sec
```

Fuzzing: Work in progress ⏳



```
── Phase 5/5: Fuzzing Report ──
============================================================
  ML-KEM Fuzzing Report
  Generated: 2026-02-14 15:09:50 UTC
  Duration:  3600s per fuzzer
  Workers:   3 fork workers per fuzzer
  Parallel:  4 concurrent fuzzers
============================================================

Fuzzer                          MaxLen   Corpus      Execs      Cov Status
------------------------------ -------- -------- ---------- -------- ------
field_arithmetic                     4        6 3874569234        5 OK
poly_ntt                           512        6  160570245       18 OK
poly_serialize                     513        9  554416958       67 OK
poly_compression                     3       14 3994164442       26 OK
poly_sampling                      193       17  470854064       44 OK
ml_kem_512_keygen                   64       24   30386461      115 OK
ml_kem_512_encaps                  832       23    9658248      270 OK
ml_kem_512_decaps                 2400       31    5463416      339 OK
ml_kem_768_keygen                   64       15   18070939      115 OK
ml_kem_768_encaps                 1216       19    5943905      253 OK
ml_kem_768_decaps                 3488       27    3568067      326 OK
ml_kem_1024_keygen                  64       43   12230654      111 OK
ml_kem_1024_encaps                1600       42    5834930      252 OK
ml_kem_1024_decaps                4736       46    3319203      327 OK

Corpus directory : /home/anjan/Documents/my_work/ml-kem/fuzz_corpus
Log files        : /home/anjan/Documents/my_work/ml-kem/fuzz_report/*.log
============================================================

[  OK]  Report saved to: /home/anjan/Documents/my_work/ml-kem/fuzz_report/summary.txt
[  OK]  No crashes or sanitizer errors found across all 14 fuzzers.

[  OK]  Total wall-clock time: 242 minutes.
```

Effective Structure Aware Fuzzing

- No need to exercise parameter validation. `std::span` ✨
- Embed knowledge of ML-KEM keys and ciphertext
- Must supply initial corpus
- Ensure corpus size grows
- Measure code-coverage
- Use `-fork=$(nproc)` for better coordination among fuzzer threads

How do I use `ml-kem` in my project?

1) Use git repository as `CMake` dependency
2) Write program, `#include “ml_kem/ml_kem_768.hpp”`
3) Build and execute

Do not want `Cmake`? Go solo. It’s just a couple of header files.
Just do `-I ml-kem/include`.

It’s all open-source and liberally licensed.

Dependency

```
include(FetchContent)
FetchContent_Declare(
  ml-kem
  GIT_REPOSITORY https://github.com/itzmeanjan/ml-kem.git
  GIT_TAG master
  GIT_SHALLOW TRUE
)
FetchContent_MakeAvailable(ml-kem)

add_executable(my_app main.cpp)
target_link_libraries(my_app PRIVATE ml-kem)
```

Btw, do you need randomness?

Use `ml-kem` header-only lib.

```
299     randomshake::randomshake_t<128> csprng;
300
301     csprng.generate(d);
302     csprng.generate(z);
303     csprng.generate(m);
304
305     ml_kem_512::keygen(d, z, pkey, skey);
306     assert(ml_kem_512::encapsulate(m, pkey, cipher, sender_key));
307     ml_kem_512::decapsulate(skey, cipher, receiver_key);
308
309     assert(sender_key == receiver_key);
310     return 0;
311   }
```

Then use `randomshake`

- Yet another cryptographically secure pseudo-random bit sequence generator.
- But C++ `<random>` header friendly.
- Built on top of (Turbo)SHAKE256 xof.
- Convenient seeding from `std::random_device`.
- It’s header-only.
- Super easy to use with C++ STL statistical distributions.

For example, generate uniformly distributed `ints` in some range

```
9   {
10    std::array<uint8_t, randomshake::randomshake_t<>::seed_byte_len> seed{};
11    seed.fill(0xfe);
12
13    // Deterministic CSPRNG : Seed -based initialization of CSPRNG
14    randomshake::randomshake_t csprng(seed);
15    std::uniform_int_distribution<uint8_t> dist{ 97, 102 };
16
17    const auto rnd_val [[maybe_unused]] = dist(csprng);
18    return EXIT_SUCCESS;
19  }
```

Curious about `randomshake`?

Technical report on eprint coming soon ⚒️

Ok, show me numbers

`ml-kem-768` on Intel x86_64 + Linux + GCC-15

ML-KEM-768 Operation	`mlkem-native`	`ml-kem`	Gap
keygen	8µs	18µs	2.25x
encaps	8µs	21µs	2.63x
decaps	11µs	26µs	2.36x



`-O3 -fomit-frame-pointer -mavx2 -mbmi2`	`-O3 -march=native`
`C90 + Hand-written AVX2 assembly for NTT + 4x parallel keccak`	Portable C++20

How is it on `aarch64`?

`ml-kem-768` on AWS Graviton4 + Linux + GCC-15

ML-KEM-768 Operation	`mlkem-native`	`ml-kem`	Gap
keygen	10µs	29µs	2.9x
encaps	11µs	33µs	3.0x
decaps	15µs	39µs	2.6x

What about `ml-dsa`?

`ml-dsa-65` on Intel x86_64 + Linux + GCC-15

ML-DSA-65 Operation	`mldsa-native`	`ml-dsa`	Gap
keygen	29µs	80µs	2.75x
sign	110µs	520µs	4.72x
verify	29µs	83µs	2.86x

Fiat-Shamir with abort?

`-O3 -fomit-frame-pointer -mavx2 -mbmi2`	`-O3 -march=native`
`C90 + Hand-written AVX2 assembly for NTT + 4x parallel keccak`	Portable C++20

How does `ml-dsa` perform on `aarch64`?

`ml-dsa-65` on AWS Graviton4 + Linux + GCC-15

ML-DSA-65 Operation	`mldsa-native`	`ml-dsa`	Gap
keygen	42µs	133µs	3.2x
sign	158µs	860µs	5.4x 
verify	41µs	136µs	3.3x

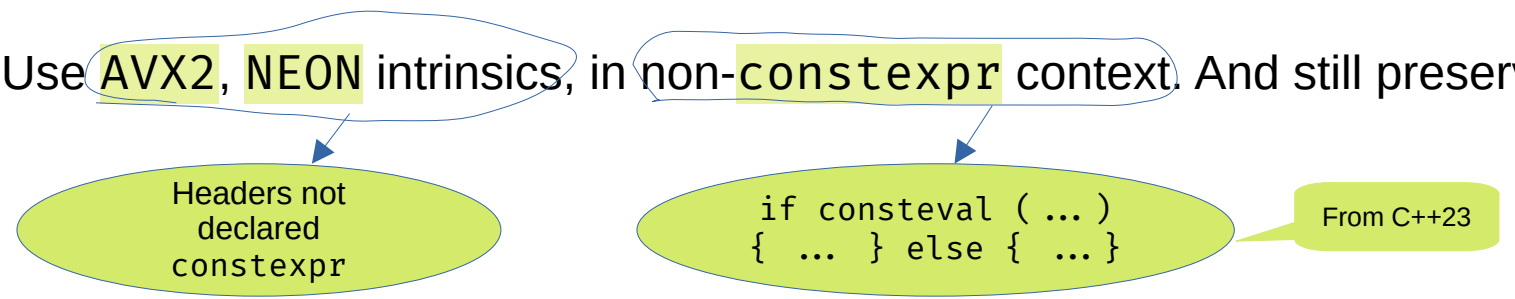
So, can they become faster?

Yes.

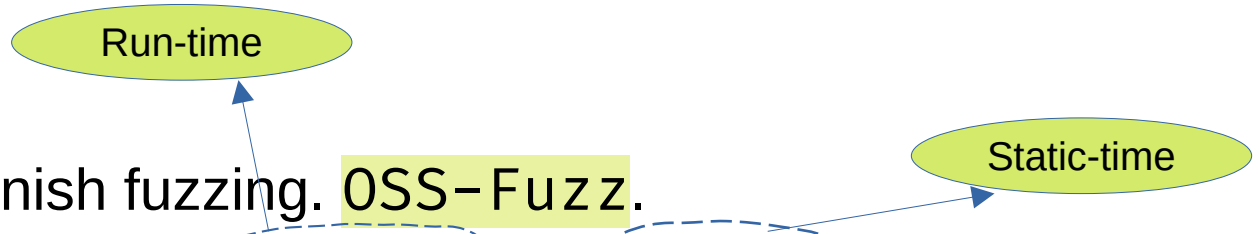
There is at least 3x room for growth 🚀 for `ml-kem` and `ml-dsa`.

How?

Use `AVX2`, `NEON` intrinsics, in non-`constexpr` context. And still preserve `constexpr` 🤩

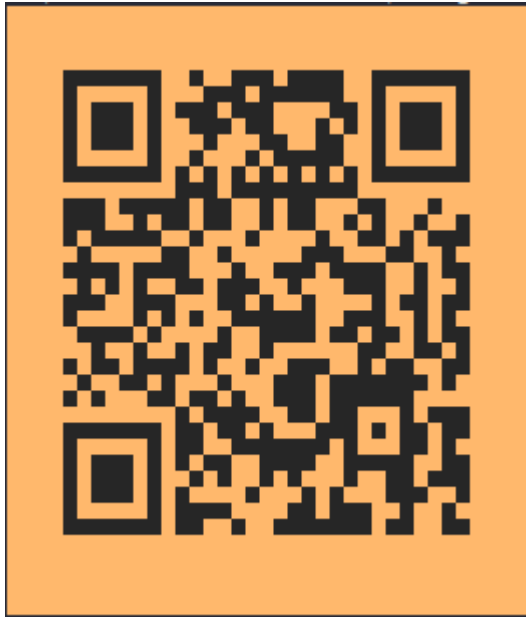


Future plans?

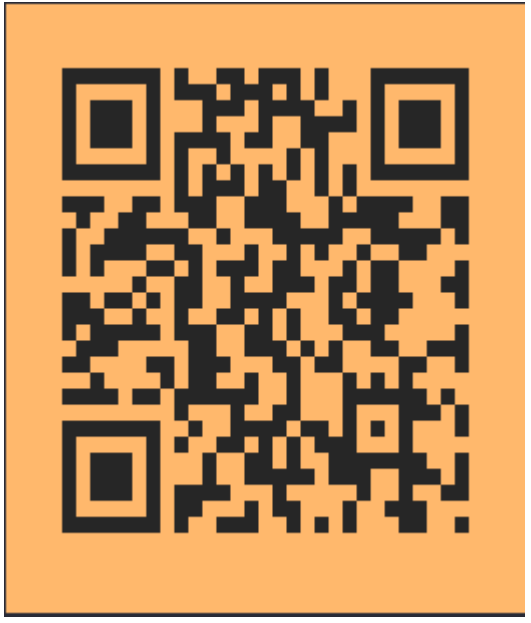


- Finish fuzzing. `OSS-Fuzz`.
- Integrate `valgrind` and `binsec` - better constant-time guarantees.
- Formal verification. Possible?
- Run faster. AVX2. NEON. RISC-V?
- Get included into PQ Code Package.
- Keep adding “useful” features. Stay lean.

That’s all for the day.
Thank you.



`ml-kem`



`ml-dsa`