A Case for Adopting Property-based Testing in Cryptographic Software Libraries

GelreCrypt 2025
Nijmegen, The Netherlands

I’m Anjan Roy,
a Software Engineer with knack for
Applied Cryptography.

Ex-Polygon, Ex-TII, PhD under Joan Daemen

What to test in crypto. software?

- Functioning as expected? Edge cases?
- Conforming to standard / spec.?
- Constant-time?
- Zeroize internal buffers with secrets?

Are there any gaps?

- How to ensure that we are actually covering all the edge cases?
- Many non-standard crypto. proposal comes with almost no test vectors. How to ensure conformance?

```rust
    #[test]
    fn kats() {
        let poseidon2 = Poseidon2::new(&POSEIDON2_GOLDILOCKS_12_PARAMS);
        let mut input: Vec<Scalar> = vec![];
        for i in 0..poseidon2.params.t {
            input.push(Scalar::from(i as u64));
        }
        let perm = poseidon2.permutation(&input);
        assert_eq!(perm[0], from_hex("0x01eaef96bdf1c0c1"));
        assert_eq!(perm[1], from_hex("0x1f0d2cc525b2540c"));
        assert_eq!(perm[2], from_hex("0x6282c1dfe1e0358d"));
        assert_eq!(perm[3], from_hex("0xe780d721f698e1e6"));
        assert_eq!(perm[4], from_hex("0x280c0b6f753d833b"));
        assert_eq!(perm[5], from_hex("0x1b942dd5023156ab"));
        assert_eq!(perm[6], from_hex("0x43f0df3fcccb8398"));
        assert_eq!(perm[7], from_hex("0xe8e8190585489025"));
        assert_eq!(perm[8], from_hex("0x56bdbf72f77ada22"));
        assert_eq!(perm[9], from_hex("0x7911c32bf9dcd705"));
        assert_eq!(perm[10], from_hex("0xec467926508fbe67"));
        assert_eq!(perm[11], from_hex("0x6a50450ddf85a6ed"));
    }
}
```

Most commonly seen number of KATs = 1

4. Test vectors

Test vectors are based on the repetition of the pattern `00 01 .. FA` with a specific length. ptn(n) defines a string by repeating the pattern `00 01 .. FA` as many times as necessary and truncated to n bytes e.g.

```
Pattern for a length of 17 bytes:
ptn(17) =
  `00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10`
```

```
Pattern for a length of 17**2 bytes:
ptn(17**2) =
  `00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
   10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
   20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
   30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
   40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
   50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
   60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
   70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
   80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
   90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
   A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
   B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
   C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
   D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
   E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
   F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA
   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
   10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
   20 21 22 23 24 25`
```

```
TurboSHAKE128(M=`00`^0, D=`07`, 32):
  `5A 22 3A D3 0B 3B 8C 66 A2 43 04 8C FC ED 43 0F
   54 E7 52 92 87 D1 51 50 B9 73 13 3A DF AC 6A 2F`

TurboSHAKE128(M=`00`^0, D=`07`, 64):
  `5A 22 3A D3 0B 3B 8C 66 A2 43 04 8C FC ED 43 0F
   54 E7 52 92 87 D1 51 50 B9 73 13 3A DF AC 6A 2F
   FE 27 08 E7 30 61 E0 9A 40 00 16 8B A9 C8 CA 18
   13 19 8F 7B BE D4 98 4B 41 85 F2 C2 58 0E E6 23`

TurboSHAKE128(M=`00`^0, D=`07`, 10032), last 32 bytes:
  `75 93 A2 80 20 A3 C4 AE 0D 60 5F D6 1F 5E B5 6E
   CC D2 7C C3 D1 2F F0 9F 78 36 97 72 A4 60 C5 5D`

TurboSHAKE128(M=ptn(1 bytes), D=`07`, 32):
  `1A C2 D4 50 FC 3B 42 05 D1 9D A7 BF CA 1B 37 51
   3C 08 03 57 7A C7 16 7F 06 FE 2C E1 F0 EF 39 E5`

TurboSHAKE128(M=ptn(17 bytes), D=`07`, 32):
  `AC BD 4A A5 75 07 04 3B CE E5 5A D3 F4 85 04 D8
   15 E7 07 FE 82 EE 3D AD 6D 58 52 C8 92 0B 90 5E`

TurboSHAKE128(M=ptn(17**2 bytes), D=`07`, 32):
  `7A 4D E8 B1 D9 27 A6 82 B9 29 61 01 03 F0 E9 64
   55 9B D7 45 42 CF AD 74 0E E3 D9 B0 36 46 9E 0A`

TurboSHAKE128(M=ptn(17**3 bytes), D=`07`, 32):
  `74 52 ED 0E D8 60 AA 8F E8 E7 96 99 EC E3 24 F8
   D9 32 71 46 36 10 DA 76 80 1E BC EE 4F CA FE 42`

TurboSHAKE128(M=ptn(17**4 bytes), D=`07`, 32):
  `CA 5F 1F 3E EA C9 92 CD C2 AB EB CA 0E 21 67 65
   DB F7 79 C3 C1 09 46 05 5A 94 AB 32 72 57 35 22`
```

Table of Contents ▲

Rare to find well crafted KATs

What can we do?

- Adopt **property-based testing (PBT)**.
- Random samples inputs, flips random bits and asserts if high-level invariants still hold.
- Better probabilistic guarantees than unit-testing+small set of KATs.
- Not as extensive or costly as fuzzing. Test run finishes faster.

PBT = Pseudo-Randomness + Repeated Property Checks

Testing AEAD

The most common pattern seen.

A fixed key, nonce, plaintext and associated data.

Replicate couple of times. And cover edges. Block length boundary.

```rust

const KEY: &[u8; 32] = &[
    0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87, 0x88, 0x89, 0x8a, 0x8b, 0x8c, 0x8d, 0x8e, 0x8f,
    0x90, 0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97, 0x98, 0x99, 0x9a, 0x9b, 0x9c, 0x9d, 0x9e, 0x9f,
];

const AAD: &[u8; 12] = &[
    0x50, 0x51, 0x52, 0x53, 0xc0, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5, 0xc6, 0xc7,
];

const PLAINTEXT: &[u8] = b"Ladies and Gentlemen of the class of '99: \
    If I could offer you only one tip for the future, sunscreen would be it.";

/// ChaCha20Poly1305 test vectors.
///
/// From RFC 8439 Section 2.8.2:
/// <https://tools.ietf.org/html/rfc8439#section-2.8.2>
mod chacha20 {
    use super::{AAD, KEY, PLAINTEXT};
    use chacha20poly1305::ChaCha20Poly1305;
    use chacha20poly1305::aead::array::Array;
    use chacha20poly1305::aead::{Aead, KeyInit, Payload};

    const NONCE: &[u8; 12] = &[
        0x07, 0x00, 0x00, 0x00, 0x40, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47,
    ];

    const CIPHERTEXT: &[u8] = &[
        0xd3, 0x1a, 0x8d, 0x34, 0x64, 0x8e, 0x60, 0xdb, 0x7b, 0x86, 0xaf, 0xbc, 0x53, 0xef, 0x7e,
        0xc2, 0xa4, 0xad, 0xed, 0x51, 0x29, 0x6e, 0x08, 0xfe, 0xa9, 0xe2, 0xb5, 0xa7, 0x36, 0xee,
        0x62, 0xd6, 0x3d, 0xbe, 0xa4, 0x5e, 0x8c, 0xa9, 0x67, 0x12, 0x82, 0xfa, 0xfb, 0x69, 0xda,
        0x92, 0x72, 0x8b, 0x1a, 0x71, 0xde, 0x0a, 0x9e, 0x06, 0x0b, 0x29, 0x05, 0xd6, 0xa5, 0xb6,
        0x7e, 0xcd, 0x3b, 0x36, 0x92, 0xdd, 0xbd, 0x7f, 0x2d, 0x77, 0x8b, 0x8c, 0x98, 0x03, 0xae,
        0xe3, 0x28, 0x09, 0x1b, 0x58, 0xfa, 0xb3, 0x24, 0xe4, 0xfa, 0xd6, 0x75, 0x94, 0x55, 0x85,
        0x80, 0x8b, 0x48, 0x31, 0xd7, 0xbc, 0x3f, 0xf4, 0xde, 0xf0, 0x8e, 0x4b, 0x7a, 0x9d, 0xe5,
        0x76, 0xd2, 0x65, 0x86, 0xce, 0xc6, 0x4b, 0x61, 0x16,
    ];

    const TAG: &[u8] = &[
        0x1a, 0xe1, 0x0b, 0x59, 0x4f, 0x09, 0xe2, 0x6a, 0x7e, 0x90, 0x2e, 0xcb, 0xd0, 0x60, 0x06,
        0x91,
    ];

    impl_tests!(
        ChaCha20Poly1305,
        KEY,
        NONCE,
        AAD,
        PLAINTEXT,
        CIPHERTEXT,
        TAG
    );

    #[test]
    fn clone_impl() {
        let _ = ChaCha20Poly1305::new(KEY.into()).clone();
    }
}
```

PBT: AEAD Edition

anjan@linux:~/Documents/my_work/ascon

```cpp

static void
test_decryption_failure_for_ascon_aead128(const size_t associated_data_len, const size_t plaintext_len, const aead_mutation_kind_t mutation_kind)
{
  EXPECT_GT(associated_data_len, 0);
  EXPECT_GT(plaintext_len, 0);

  std::array<uint8_t, ascon_aead128::KEY_BYTE_LEN> key{};
  std::array<uint8_t, ascon_aead128::NONCE_BYTE_LEN> nonce{};
  std::array<uint8_t, ascon_aead128::TAG_BYTE_LEN> tag{};
  std::vector<uint8_t> associated_data(associated_data_len, 0);
  std::vector<uint8_t> plaintext(plaintext_len, 0);
  std::vector<uint8_t> ciphertext(plaintext_len, 0);
  std::vector<uint8_t> decipheredtext(plaintext_len, 0xff);

  generate_random_data<uint8_t>(key);
  generate_random_data<uint8_t>(nonce);
  generate_random_data<uint8_t>(associated_data);
  generate_random_data<uint8_t>(plaintext);

  ascon_aead128::encrypt(key, nonce, associated_data, plaintext, ciphertext, tag);

  switch (mutation_kind) {
    case aead_mutation_kind_t::mutate_key:
      do_bitflip(key);
      break;
    case aead_mutation_kind_t::mutate_nonce:
      do_bitflip(nonce);
      break;
    case aead_mutation_kind_t::mutate_tag:
      do_bitflip(tag);
      break;
    case aead_mutation_kind_t::mutate_associated_data:
      do_bitflip(associated_data);
      break;
    case aead_mutation_kind_t::mutate_cipher_text:
      do_bitflip(ciphertext);
      break;
    default:
      EXPECT_TRUE(false);
  }

  const auto is_decrypted = ascon_aead128::decrypt(key, nonce, associated_data, ciphertext, decipheredtext, tag);
  EXPECT_FALSE(is_decrypted);

  std::vector<uint8_t> zeros(plaintext_len, 0);
  EXPECT_EQ(decipheredtext, zeros);
}

```

1 Random sample key, nonce, associated data and plaintext

2 Random bit-flip

3 Assert decryption failure+zeroize decipheredtext buffer

Testing Hash Fn.

```
test "SHAKE-128 single" {
    var out: [10]u8 = undefined;
    Shake128.hash("hello123", &out, .{});
    try htest.assertEqual("1b85861510bc4d8e467d", &out);
}

test "SHAKE-128 multisqueeze" {
    var out: [10]u8 = undefined;
    var h = Shake128.init(.{});
    h.update("hello123");
    h.squeeze(out[0..4]);
    h.squeeze(out[4..]);
    try htest.assertEqual("1b85861510bc4d8e467d", &out);
}

test "SHAKE-128 multisqueeze with multiple blocks" {
    var out: [100]u8 = undefined;
    var out2: [100]u8 = undefined;

    var h = Shake128.init(.{});
    h.update("hello123");
    h.squeeze(out[0..50]);
    h.squeeze(out[50..]);

    var h2 = Shake128.init(.{});
    h2.update("hello123");
    h2.squeeze(&out2);
    try std.testing.expectEqualSlices(u8, &out, &out2);
}
```

PBT: Hash Edition

Absorb all input at once 1

Incremental randomized absorption, by building look-ahead buffer 2

Assert digest equality 3

anjan@linux:~/Documents/my_work/ascon

```cpp
TEST(AsconHash256, ForSameMessageOneshotHashingAndIncrementalHashingProducesSameDigest)
{
  for (size_t msg_byte_len = MIN_MSG_LEN; msg_byte_len <= MAX_MSG_LEN; msg_byte_len++) {
    std::array<uint8_t, ascon_hash256::DIGEST_BYTE_LEN> digest_oneshot{};
    std::array<uint8_t, ascon_hash256::DIGEST_BYTE_LEN> digest_multishot{};

    digest_oneshot.fill(0x5f);
    digest_oneshot.fill(0x3f);

    std::vector<uint8_t> msg(msg_byte_len);
    auto msg_span = std::span(msg);

    generate_random_data<uint8_t>(msg_span);

    // Oneshot hashing
    {
      ascon_hash256::ascon_hash256_t hasher;

      EXPECT_TRUE(hasher.absorb(msg_span));
      EXPECT_TRUE(hasher.finalize());
      EXPECT_TRUE(hasher.digest(digest_oneshot));
    }

    // Incremental hashing
    {
      ascon_hash256::ascon_hash256_t hasher;

      size_t msg_offset = 0;
      while (msg_offset < msg_byte_len) {
        // Because we don't want to be stuck in an infinite loop if msg[msg_offset] = 0
        const auto elen = std::min<size_t>(std::max<uint8_t>(msg[msg_offset], 1), msg_byte_len - msg_offset);

        EXPECT_TRUE(hasher.absorb(msg_span.subspan(msg_offset, elen)));
        msg_offset += elen;
      }

      EXPECT_TRUE(hasher.finalize());
      EXPECT_TRUE(hasher.digest(digest_multishot));
    }

    EXPECT_EQ(digest_oneshot, digest_multishot);
  }
}
```

(END)

PBT: XOF Edition

Absorb all input at once, then squeeze it all out in a single go.

1

Absorb input in pseudo-random order. Depends on message bytes itself.

2

Squeeze output in pseudo-random order. Depends on output bytes itself.

3

anjan@linux:~/Documents/my_work/ascon

```
TEST(AsconXof128, ForSameMessageOneshotHashingAndIncrementalHashingProducesSameOutput)
{
  for (size_t msg_byte_len = MIN_MSG_LEN; msg_byte_len <= MAX_MSG_LEN; msg_byte_len++) {
    for (size_t output_byte_len = MIN_OUT_LEN; output_byte_len <= MAX_OUT_LEN; output_byte_len++) {
      std::vector<uint8_t> msg(msg_byte_len);
      std::vector<uint8_t> digest_oneshot(output_byte_len, 0x5f);
      std::vector<uint8_t> digest_multishot(output_byte_len, 0x3f);

      auto msg_span = std::span(msg);
      auto digest_oneshot_span = std::span(digest_oneshot);
      auto digest_multishot_span = std::span(digest_multishot);

      generate_random_data(msg_span);

      // Oneshot hashing
      {
        ascon_xof128::ascon_xof128_t hasher;

        EXPECT_EQ(hasher.absorb(msg_span), ascon_xof128::ascon_xof128_status_t::absorbed_data);
        EXPECT_EQ(hasher.finalize(), ascon_xof128::ascon_xof128_status_t::finalized_data_absorption_phase);
        EXPECT_EQ(hasher.squeeze(digest_oneshot_span), ascon_xof128::ascon_xof128_status_t::squeezed_output);
      }

      // Incremental hashing
      {
        ascon_xof128::ascon_xof128_t hasher;

        size_t msg_offset = 0;
        while (msg_offset < msg_byte_len) {
          // Because we don't want to be stuck in an infinite loop if msg[off] = 0
          const auto elen = std::min<size_t>(std::max<uint8_t>(msg_span[msg_offset], 1), msg_byte_len - msg_offset);

          EXPECT_EQ(hasher.absorb(msg_span.subspan(msg_offset, elen)), ascon_xof128::ascon_xof128_status_t::absorbed_data);
          msg_offset += elen;
        }

        EXPECT_EQ(hasher.finalize(), ascon_xof128::ascon_xof128_status_t::finalized_data_absorption_phase);

        // Squeeze message bytes in many iterations
        size_t output_offset = 0;
        while (output_offset < output_byte_len) {
          EXPECT_EQ(hasher.squeeze(digest_multishot_span.subspan(output_offset, 1)), ascon_xof128::ascon_xof128_status_t::squeezed_output);

          auto elen = std::min<size_t>(digest_multishot_span[output_offset], output_byte_len - (output_offset + 1));

          output_offset += 1;
          EXPECT_EQ(hasher.squeeze(digest_multishot_span.subspan(output_offset, elen)), ascon_xof128::ascon_xof128_status_t::squeezed_output);
          output_offset += elen;
        }
      }

      EXPECT_EQ(digest_oneshot, digest_multishot);
    }
  }
}
```

Testing ML-KEM

1) Generate keypair from seed.
2) Encapsulate, get ciphertext and `sharedSecret`.
3) Decapsulate, get back `sharedSecret’`.
4) Assert `sharedSecret = sharedSecret’`.

```

test "Test happy flow" {
    var seed: [64]u8 = undefined;
    for (&seed, 0..) |*s, i| {
        s.* = @as(u8, @intCast(i));
    }
    inline for (modes) |mode| {
        for (0..10) |i| {
            seed[0] = @as(u8, @intCast(i));
            const kp = try mode.KeyPair.generateDeterministic(seed);
            const sk = try mode.SecretKey.fromBytes(&kp.secret_key.toBytes());
            try testing.expectEqual(sk, kp.secret_key);
            const pk = try mode.PublicKey.fromBytes(&kp.public_key.toBytes());
            try testing.expectEqual(pk, kp.public_key);
            for (0..10) |j| {
                seed[1] = @as(u8, @intCast(j));
                const e = pk.encaps(seed[0..32].*);
                try testing.expectEqual(e.shared_secret, try sk.decaps(&e.ciphertext));
            }
        }
    }
}
```

PBT: ML-KEM Edition (1)

anjan@linux:~/Documents/my_work/ml-kem

```cpp
// For ML-KEM-768
//
// - Generate a valid keypair.
// - Malform public key s.t. last coefficient of polynomial vector is not properly reduced.
// - Attempt to encapsulate using malformed public key. It must fail.
TEST(ML_KEM, ML_KEM_768_EncapsFailureDueToNonReducedPubKey)
{
  std::array<uint8_t, ml_kem_768::SEED_D_BYTE_LEN> seed_d{};
  std::array<uint8_t, ml_kem_768::SEED_Z_BYTE_LEN> seed_z{};
  std::array<uint8_t, ml_kem_768::SEED_M_BYTE_LEN> seed_m{};

  std::array<uint8_t, ml_kem_768::PKEY_BYTE_LEN> pubkey{};
  std::array<uint8_t, ml_kem_768::SKEY_BYTE_LEN> seckey{};
  std::array<uint8_t, ml_kem_768::CIPHER_TEXT_BYTE_LEN> cipher{};

  std::array<uint8_t, ml_kem_768::SHARED_SECRET_BYTE_LEN> shared_secret{};

  randomshake::randomshake_t<192> csprng{};
  csprng.generate(seed_d);
  csprng.generate(seed_z);
  csprng.generate(seed_m);

  ml_kem_768::keygen(seed_d, seed_z, pubkey, seckey);

  make_malformed_pubkey<pubkey.size()>(pubkey);
  const auto is_encapsulated = ml_kem_768::encapsulate(seed_m, pubkey, cipher, shared_secret);

  EXPECT_FALSE(is_encapsulated);
}
```

Malform public key, assert encaps failing anjan@linux:~/Documents/my_work/ml-kem

PBT: ML-KEM Edition (2)

```cpp
// For ML-KEM-768
//
// - Generate a valid keypair.
// - Encapsulate using public key, generate shared secret, at sender's side.
// - Cause a random bitflip in cipher text, at receiver's side.
// - Attempt to decapsulate bit-flipped cipher text, using valid secret key. Must fail *implicitly*.
// - Shared secret of sender and receiver must not match.
// - Shared secret at receiver's end must match `seed_z`, which is last 32 -bytes of secret key.
TEST(ML_KEM, ML_KEM_768_DecapsFailureDueToBitFlippedCipherText)
{
  std::array<uint8_t, ml_kem_768::SEED_D_BYTE_LEN> seed_d{};
  std::array<uint8_t, ml_kem_768::SEED_Z_BYTE_LEN> seed_z{};
  std::array<uint8_t, ml_kem_768::SEED_M_BYTE_LEN> seed_m{};

  std::array<uint8_t, ml_kem_768::PKEY_BYTE_LEN> pubkey{};
  std::array<uint8_t, ml_kem_768::SKEY_BYTE_LEN> seckey{};
  std::array<uint8_t, ml_kem_768::CIPHER_TEXT_BYTE_LEN> cipher{};

  std::array<uint8_t, ml_kem_768::SHARED_SECRET_BYTE_LEN> shared_secret_sender{};
  std::array<uint8_t, ml_kem_768::SHARED_SECRET_BYTE_LEN> shared_secret_receiver{};

  randomshake::randomshake_t<192> csprng{};
  csprng.generate(seed_d);
  csprng.generate(seed_z);
  csprng.generate(seed_m);

  ml_kem_768::keygen(seed_d, seed_z, pubkey, seckey);
  const auto is_encapsulated = ml_kem_768::encapsulate(seed_m, pubkey, cipher, shared_secret_sender);

  random_bitflip_in_cipher_text<cipher.size()>(cipher, csprng);
  ml_kem_768::decapsulate(seckey, cipher, shared_secret_receiver);

  EXPECT_TRUE(is_encapsulated);
  EXPECT_NE(shared_secret_sender, shared_secret_receiver);
  EXPECT_EQ(shared_secret_receiver, seed_z);
  EXPECT_TRUE(std::equal(shared_secret_receiver.begin(), shared_secret_receiver.end(), std::span(seckey).last<32>().begin()));
}
```

(END)

Random bit-flip ciphertext

Assert diff. shared-secret at two ends

Testing Galois Field Arithmetic Part (1)

1) Test with additive or multiplicative identity.
2) Test with result overflowing.

```
#[test]
fn mul() {
    // identity
    let r: BaseElement = rand_value();
    assert_eq!(BaseElement::ZERO, r * BaseElement::ZERO);
    assert_eq!(r, r * BaseElement::ONE);

    // test multiplication within bounds
    assert_eq!(BaseElement::from(15u8), BaseElement::from(5u8) * BaseElement::from(3u8));

    // test overflow
    let m = BaseElement::MODULUS;
    let t = BaseElement::new(m - 1);
    assert_eq!(BaseElement::ONE, t * t);
    assert_eq!(BaseElement::new(m - 2), t * BaseElement::from(2u8));
    assert_eq!(BaseElement::new(m - 4), t * BaseElement::from(4u8));

    #[allow(clippy::manual_div_ceil)]
    let t = (m + 1) / 2;
    assert_eq!(BaseElement::ONE, BaseElement::new(t) * BaseElement::from(2u8));
}
```

Testing Galois Field Arithmetic Part (2)

1) Test with additive or multiplicative identity.
2) Direct exponentiate vs. Exponentiate by repeated mult.

```rust
#[test]
fn exp() {
    let a = BaseElement::ZERO;
    assert_eq!(a.exp(0), BaseElement::ONE);
    assert_eq!(a.exp(1), BaseElement::ZERO);
    assert_eq!(a.exp7(), BaseElement::ZERO);

    let a = BaseElement::ONE;
    assert_eq!(a.exp(0), BaseElement::ONE);
    assert_eq!(a.exp(1), BaseElement::ONE);
    assert_eq!(a.exp(3), BaseElement::ONE);
    assert_eq!(a.exp7(), BaseElement::ONE);

    let a: BaseElement = rand_value();
    assert_eq!(a.exp(3), a * a * a);
    assert_eq!(a.exp(7), a.exp7());
}
```

PBT: GF Edition

Random sample a pair of GF elems

Assert: round-trip testing for +, -

Assert: round-trip testing for x, /

<u>Metamorphism</u>

anjan@linux:~/Documents/my_work/ml-kem

File: tests/test_field.cpp

```cpp
#include "ml_kem/internals/math/field.hpp"
#include "randomshake/randomshake.hpp"
#include <gtest/gtest.h>

// Test functional correctness of ML-KEM prime field operations, by running through multiple rounds
// of execution of field operations on randomly sampled field elements.
TEST(ML_KEM, ArithmeticOverZq)
{
  constexpr size_t ITERATION_COUNT = 1ul << 20;

  randomshake::randomshake_t<128> csprng{};

  for (size_t i = 0; i < ITERATION_COUNT; i++) {
    const auto a = ml_kem_field::zq_t::random(csprng);
    const auto b = ml_kem_field::zq_t::random(csprng);

    // Addition, Subtraction and Negation
    const auto c = a + b;
    const auto d = c - b;
    const auto e = c - a;

    EXPECT_EQ(d, a);
    EXPECT_EQ(e, b);

    // Multiplication, Exponentiation, Inversion and Division
    const auto f = a * b;
    const auto g = f / b;
    const auto h = f / a;

    if (b != ml_kem_field::zq_t::zero()) {
      EXPECT_EQ(g, a);
    } else {
      EXPECT_EQ(g, ml_kem_field::zq_t());
    }

    if (a != ml_kem_field::zq_t::zero()) {
      EXPECT_EQ(h, b);
    } else {
      EXPECT_EQ(h, ml_kem_field::zq_t());
    }
  }
}
```

```
ml-kem on  ml-kem-b [?]
❯
```

Any libraries to take up PBT boilerplate?

- `proptest` and `quickcheck` for Rust.
- `hypothesis` for Python.
- `rapidcheck` for C++. It’s `quickcheck` clone.
- Roll your own minimal PBT infra, in your favourite language.

Easy PBT in Python, using `hypothesis`

anjan@linux:~/Documents/my_work/verifiable-rlnc

```python

from hypothesis import given, strategies as st, settings, assume, HealthCheck

@given(
    msg_a=st.binary(min_size=MIN_MSG_BYTE_LEN, max_size=MAX_MSG_BYTE_LEN),
    msg_b=st.binary(min_size=MIN_MSG_BYTE_LEN, max_size=MAX_MSG_BYTE_LEN),
    eval_at=st.integers(min_value=1, max_value=gfp.GFP.order - 1),
)
@settings(max_examples=10_000, derandomize=False, deadline=None, suppress_health_check=[HealthCheck.filter_too_much])
def test_additive_homomorphism_of_poly_eval_hash(msg_a: bytes, msg_b: bytes, eval_at: int):
    assume(len(msg_a) == len(msg_b))

    hash_a = ph.poly_eval_hash(msg_a, eval_at)
    hash_b = ph.poly_eval_hash(msg_b, eval_at)

    msg_ab = add_two_messages(msg_a, msg_b)
    expected_hash_ab = add_two_messages(hash_a, hash_b)

    computed_hash_ab = ph.poly_eval_hash(msg_ab, eval_at)
    assert expected_hash_ab == computed_hash_ab

:
```

Thank you for your time and attention.