CFRG Specifications in Theory and Practice

CFRG - IETF 113 - Vienna

Disclaimer

1. This is based on my personal experience accumulated in the CFRG, and does not represent the group’s shared view
2. It is not meant to point fingers or assign blame – it’s meant to highlight ways we can improve the group’s primary deliverables

CFRG specifications in theory

CFRG charter: The CFRG serves as a bridge between theory and practice, bringing new cryptographic techniques to the Internet community and promoting an understanding of the use and applicability of these mechanisms via **Informational RFCs**…

RFC2026: An "**Informational**" specification is published for the general information of the Internet community, and does not represent an Internet community consensus or recommendation. The Informational designation is … subject only to editorial considerations and to verification that there has been adequate coordination with the standards process.

CFRG specifications in theory

CFRG charter: The CFRG serves as a bridge between theory and practice, bringing new cryptographic techniques to the Internet community and promoting an understanding of the use and applicability of these mechanisms via **Informational RFCs**…

RFC2026: An "**Informational**" specification is **published for the general information of the Internet community**, and does not represent an Internet community consensus or recommendation. The Informational designation is … **subject only to editorial considerations and to verification that there has been adequate coordination with the standards process**.

CFRG specifications in practice

CFRG specifications have significant impact on protocol design, security analysis, and implementations:

- RFC2104: HMAC
- RFC5869: HKDF
- RFC7748: Curve25519/X25519
- RFC8032: EdDSA
- RFC9180: HPKE
- … any many more

CFRG specifications target a wide variety of audiences:

- Protocol designers and implementers
- Cryptographic reviewers
- …

Specification components

There are (at least) three different parts of a specification:

1. Functional specification. What does this object do? What is its purpose?
2. Syntax specification. How do I interact with this object?
3. Implementation specification. How does this object work internally?

Presentation of each should be tailored to its audience

Key questions for specification writers

1. Is the specification easy to understand and use?
    a. Is the functional description of the cryptographic object clear?
    b. What is the cognitive load required to understand the specification?
    c. Is the syntax of the object clear?
2. Will the specification yield consistent and correct implementations?
    a. Is the functional behavior well-defined?
    b. Is the implementation description clear?

Example: RFC8032 (EdDSA)

Consider RFC8032's Verify description:

1. To verify a signature on a message M using public key A, with F being 0 for Ed25519ctx, 1 for Ed25519ph, and if Ed25519ctx or Ed25519ph is being used, C being the context, first split the signature into two 32-octet halves. Decode the first half as a point R, and the second half as an integer S, in the range 0 <= s < L. Decode the public key A as point A'. If any of the decodings fail (including S being out of range), the signature is invalid.

2. Compute SHA512(dom2(F, C) || R || A || PH(M)), and interpret the 64-octet digest as a little-endian integer k.

3. Check the group equation [8][S]B = [8]R + [8][k]A'. It's sufficient, but not required, to instead check [S]B = R + [k]A'.

Questions:

- Is the specification easy to understand and use?
- Will the specification yield consistent and correct implementations?

Example: RFC8032 (EdDSA)

Consider RFC8032’s Verify description:

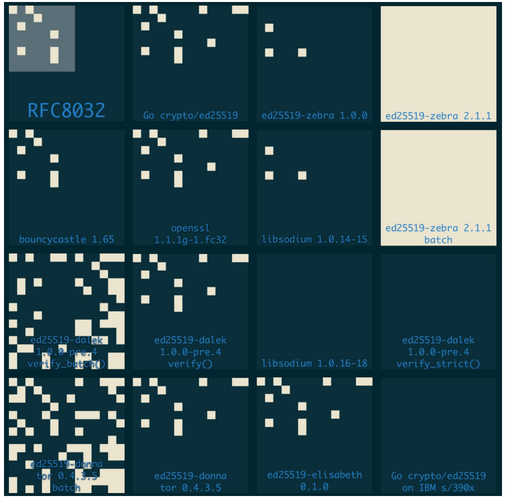
1. To verify a signature on a message M using public key A, with F being 0 for Ed25519ctx, 1 for Ed25519ph, and if Ed25519ctx or Ed25519ph is being used, C being the context, first split the signature into two 32-octet halves. Decode the first half as a point R, and the second half as an integer S, in the range 0 <= s < L. Decode the public key A as point A'. If any of the decodings fail (including S being out of range), the signature is invalid.

2. Compute SHA512(dom2(F, C) || R || A || PH(M)), and interpret the 64-octet digest as a little-endian integer k.

3. Check the group equation [8][S]B = [8]R + [8][k]A'. It's sufficient, but not required, to instead check [S]B = R + [k]A'.

Questions:

- Is the specification easy to understand and use? ❓
- Will the specification yield consistent and correct implementations? ❎



Source: https://hdevalence.ca/blog/2020-10-04-its-25519am

Example: RFC8032 (EdDSA)

Consider RFC8032’s Verify description:

1. To verify a signature on a message M using public key A, with F being 0 for Ed25519ctx, 1 for Ed25519ph, and if Ed25519ctx or Ed25519ph is being used, C being the context, first split the signature into two 32-octet halves. Decode the first half as a point R, and the second half as an integer S, in the range 0 <= s < L. Decode the public key A as point A'. If any of the decodings fail (including S being out of range), the signature is invalid.

2. Compute SHA512(dom2(F, C) || R || A || PH(M)), and interpret the 64-octet digest as a little-endian integer k.

3. Check the group equation [8][S]B = [8]R + [8][k]A'. It's sufficient, but not required, to instead check [S]B = R + [k]A'.

Questions:

- Is the specification easy to understand and use? ❓
- Will the specification yield consistent and correct implementations? ❎

Ambiguous implementation description

Source: https://hdevalence.ca/blog/2020-10-04-its-25519am

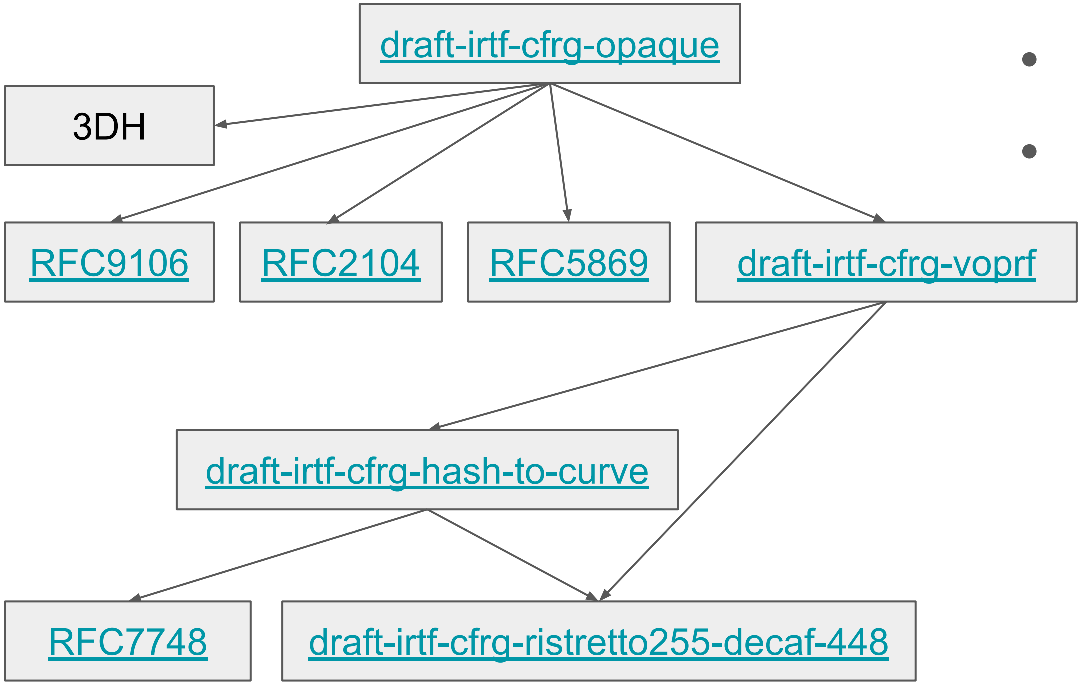
Example: OPAQUE

draft-irtf-cfrg-opaque

Questions:

- Is the specification easy to understand and use?
- Will the specification yield consistent and correct implementations?

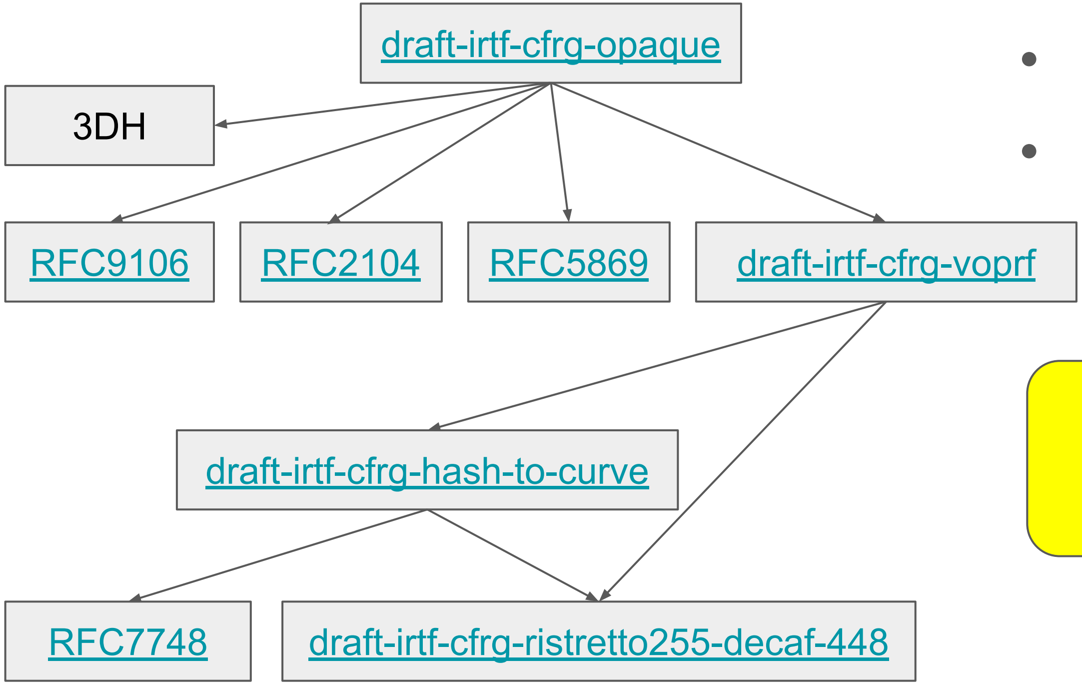
Example: OPAQUE



Questions:

- Is the specification easy to understand and use? ❎
- Will the specification yield consistent and correct implementations? ❓

Example: OPAQUE



Questions:

- Is the specification easy to understand and use? ❎
- Will the specification yield consistent and correct implementations? ❓

Huge cognitive load required to understand the entire protocol

Thesis

Problem statement:

- CFRG produces technical specifications for cryptographic objects that are consumed by a diverse audience
- Each object is expected to have easy-to-understand and well-defined behavior with clear syntax (API)
- Failure to establish this clarity and consistency will yield specifications with little or no value and possibly harmful consequences in practice

Writing technical specifications such that they detail cryptographic objects with **well-defined behavior** and **clear syntax** is challenging

Case study: `hash-to-curve`

Hash-to-curve overview

`hash_to_curve` is a uniform encoding from byte strings to points in some elliptic curve group G

```
hash_to_curve(msg)

Input: msg, an arbitrary-length byte string.
Output: P, a point in G.

Steps:
1. u = hash_to_field(msg, 2)
2. Q0 = map_to_curve(u[0])
3. Q1 = map_to_curve(u[1])
4. R = Q0 + Q1              # Point addition
5. P = clear_cofactor(R)
6. return P
```

Hash-to-curve overview

`hash_to_curve` is a uniform encoding from byte strings to points in some elliptic curve group G

```
hash_to_curve(msg)

Input: msg, an arbitrary-length byte string.
Output: P, a point in G.

Steps:
1. u = hash_to_field(msg, 2)
2. Q0 = map_to_curve(u[0])
3. Q1 = map_to_curve(u[1])
4. R = Q0 + Q1          # Point addition
5. P = clear_cofactor(R)
6. return P
```

Questions:

- Is the specification easy to understand and use?
- Will the specification yield consistent and correct implementations?

Functional specification: map-to-curve

The function map_to_curve calculates a point on the elliptic curve E from an element of the finite field F over which E is defined. Section 6 describes mappings for a range of curve families.

```
map_to_curve(u)
Input: u, an element of field F.
Output: Q, a point on the elliptic curve E.
Steps: defined in Section 6.
```

Functional description suitable for understanding what it is this object does and why

Syntax specification: map-to-curve

Preconditions: A Montgomery curve K * t^2 = s^3 + J * s^2 + s where J != 0, K != 0, and (J^2 - 4) / K^2 is non-zero and non-square in F.

Constants:
J and K, the parameters of the elliptic curve.
Z, a non-square element of F. Appendix H.3 gives a Sage [SAGE] script that outputs the RECOMMENDED Z.

Exceptions: The exceptional case is Z * u^2 == -1, i.e., 1 + Z * u^2 == 0. Implementations must detect this case and set x1 = -(J / K). Note that this can only happen when q = 3 (mod 4).

Operations:

```
1.  x1 = -(J / K) * inv0(1 + Z * u^2)
2.  If x1 == 0, set x1 = -(J / K)
3. gx1 = x1^3 + (J / K) * x1^2 + x1 / K^2
4.  x2 = -x1 - (J / K)
5. gx2 = x2^3 + (J / K) * x2^2 + x2 / K^2
6.  If is_square(gx1), set x = x1, y = sqrt(gx1) with sgn0(y) == 1.
7.  Else set x = x2, y = sqrt(gx2) with sgn0(y) == 0.
8.   s = x * K
9.   t = y * K
10. return (s, t)
```

Syntax suitable for understanding the high-level functionality

Implementation specification: map-to-curve

```
map_to_curve_elligator2(u)

Input: u, an element of F.
Output: (s, t), a point on M.

Constants:
1.   c1 = J / K
2.   c2 = 1 / K^2

Steps:
1.  tv1 = u^2
2.  tv1 = Z * tv1            # Z * u^2
3.   e1 = tv1 == -1          # exceptional case: Z * u^2 == -1
4.  tv1 = CMOV(tv1, 0, e1)   # if tv1 == -1, set tv1 = 0
5.   x1 = tv1 + 1
6.   x1 = inv0(x1)
7.   x1 = -c1 * x1           # x1 = -(J / K) / (1 + Z * u^2)
8.  gx1 = x1 + c1
9.  gx1 = gx1 * x1
10. gx1 = gx1 + c2
11. gx1 = gx1 * x1           # gx1 = x1^3 + (J / K) * x1^2 + x1 / K^2
12.  x2 = -x1 - c1
13. gx2 = tv1 * gx1
14.  e2 = is_square(gx1)     # If is_square(gx1)
15.   x = CMOV(x2, x1, e2)   #   then  x = x1,  else  x = x2
16.  y2 = CMOV(gx2, gx1, e2) #   then y2 = gx1, else y2 = gx2
17.   y = sqrt(y2)
…
```

Implementation description suitable for bridging the gap between mathematical description and required behavior

No one-size-fits-all

Delta between functional, syntax, and implementation specifications is a delicate balance

Functional specification should be maximally clear for people trying to understand what the object does without understanding *how* it works

Syntax specification should follow from the functional specification and be easy to use and hard to misuse

Implementation specification should given implementers confidence in their implementation

Achieving balance

Generally aim towards alignment between across functional, syntax, and implementation specifications

- Use consistent pseudocode to describe all three
- Make pseudocode match reference implementation as close as possible

Improve clarity by reusing concepts and notation

- Use consistent terminology and vocabulary
- Adopt consistent presentation format (e.g., for pseudocode)

Consistency and clarity across drafts

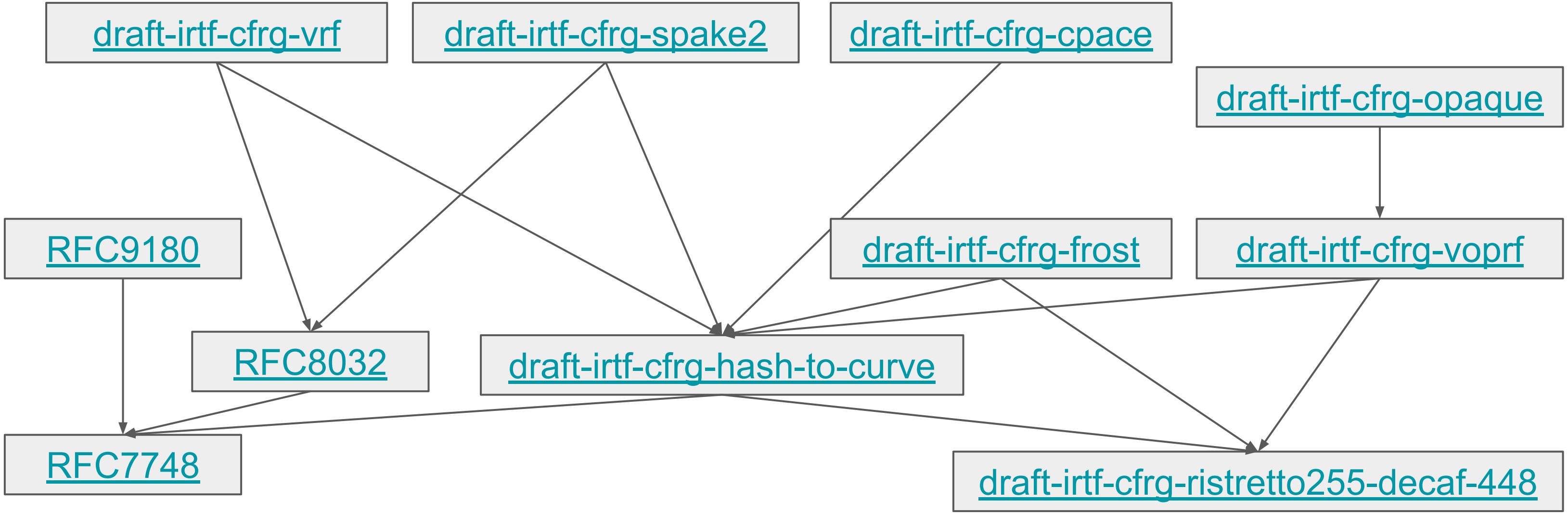
CFRG strives to produce high quality specifications of cryptographic objects

… but consistency across drafts is lacking

Consistency and clarity across drafts

CFRG strives to produce high quality specifications of cryptographic objects

… but consistency across drafts is lacking



Wrapping up

CFRG specifications (... standards …) are important to the community, and clarity is of the utmost importance for the documents produced

Most drafts are clear and internally consistent, but there's more that can be done

- Reference implementation requirements and reviews?
- Common requirements for syntax (APIs)?
- Reference implementation derived from specification a la hacspec? How can formal methods help improve specification quality?

The group lacks consistency across drafts, which is solvable problem

- Use common terminology, concepts, and notation across *related* documents?
- Share reference implementations for *related* documents?

CFRG Specifications in Theory and Practice

CFRG - IETF 113 - Vienna