CFRG Specifications
In theory and practice

Christopher A. Wood — Research Lead — Cloudflare

ANRW 2022 — IETF 114 — Philadelphia

Disclaimer: This is based on my personal experience accumulated in the CFRG, and **does not represent the group’s shared view**

Goal: Highlight ways we can improve the group’s primary deliverables, not point fingers or assign blame

What is the problem?

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

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

- RFC2104: HMAC
- RFC5869: HKDF
- RFC7748: Curve25519/X25519
- RFC8032: EdDSA
- RFC9180: HPKE

Specifications target a wide variety of audiences:

- Protocol designers and implementers
- Cryptographic reviewers
- …

CFRG Specifications

In practice

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

- RFC2104: HMAC
- RFC5869: HKDF
- RFC7748: Curve25519/X25519
- RFC8032: EdDSA
- RFC9180: HPKE

Specifications target a wide variety of audiences:

- Protocol designers and implementers
- Cryptographic reviewers
- …

CFRG Specifications

In practice

Specifications have significant impact on protocol design, security analysis and implem

- RFC2104:
- RFC5869:
- RFC7748:
- RFC8032: EdDSA
- RFC9180: HPKE

Specifications target a wide variety of audiences:

- menters
- …

High stakes

Quality is critical

Specification quality

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

Guiding Questions

1. Is the specification **easy to understand and use**?

   Is the functional description of the cryptographic object clear?

   What is the cognitive load required to understand the specification?

   Is the syntax of the object clear?

2. Will the specification yield **consistent and correct implementations**?

   Is the functional behavior well-defined?

   Is the syntax correct and amenable to type-safe implementations?

   Is the implementation description clear?

Example: RFC8032 (EdDSA)

```
5.1.7.  Verify

   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'.
```

1. Is the specification easy to understand and use?

2. Will the specification yield consistent and correct implementations?

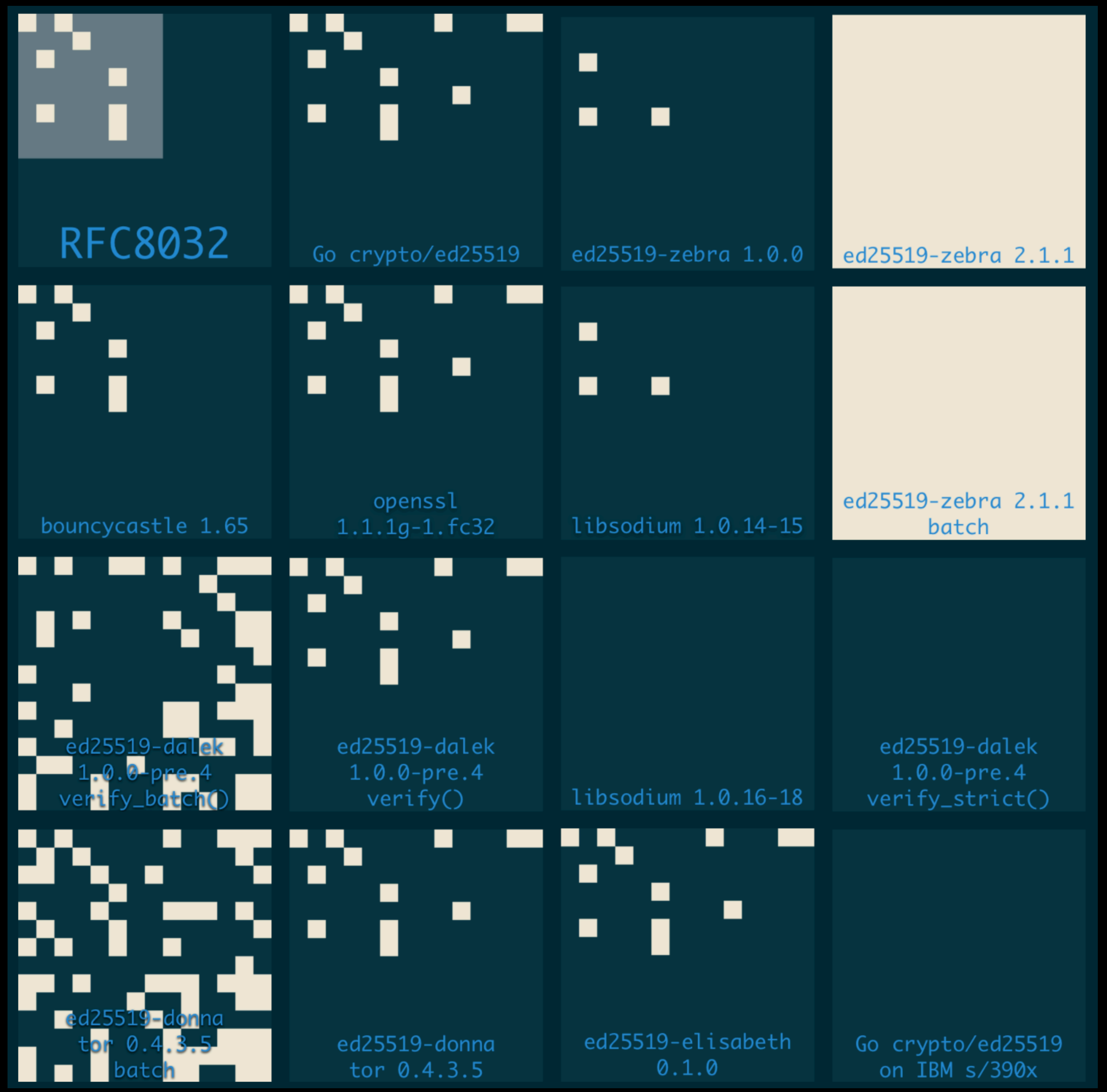
Example: RFC8032 (EdDSA)

5.1.7. Verify

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'.



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

1. Is the specification easy to understand and use? ✔

2. Will the specification yield consistent and correct implementations? ✘

Example: OPAQUE (draft-irtf-cfrg-opaque)

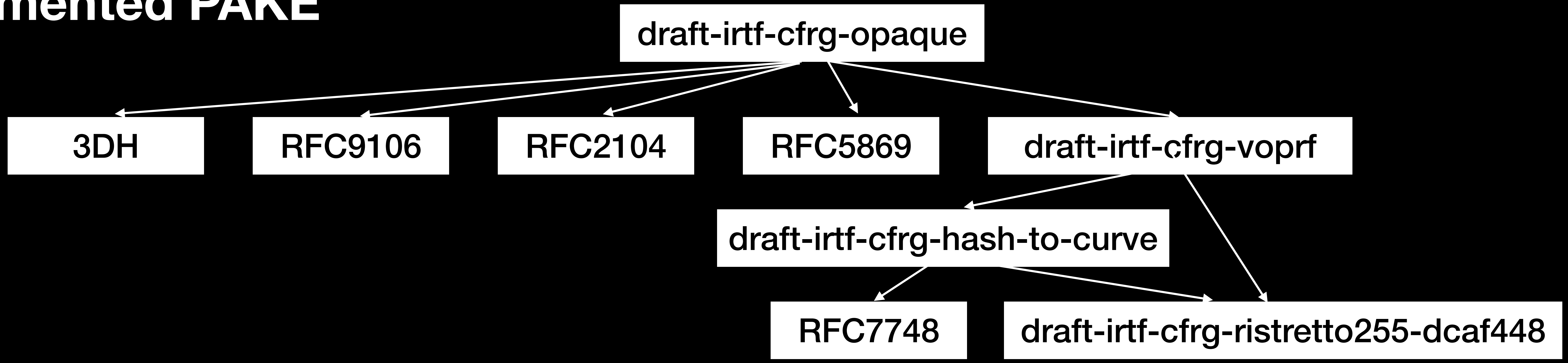
OPAQUE is a compiler for translating an OPRF, hash function, memory hard function (MHF), and authenticated key exchange (AKE) protocol into a **strong, augmented PAKE**

1. Is the specification easy to understand and use?

2. Will the specification yield consistent and correct implementations?

Example: OPAQUE (draft-irtf-cfrg-opaque)

OPAQUE is a compiler for translating an OPRF, hash function, memory hard function (MHF), and authenticated key exchange (AKE) protocol into a **strong, augmented PAKE**



1. Is the specification easy to understand and use? ✗

2. Will the specification yield consistent and correct implementations? ✓

Core Problem

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
```

1. Is the specification easy to understand and use?

2. Will the specification yield consistent and correct implementations?

For the specification user…

Functional description

```
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
```

For the specification user…

Functional description

```
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_field(msg, count)

Inputs:
- msg, a byte string containing the message to hash.
- count, the number of elements of F to output.

Outputs:
- (u_0, ..., u_(count - 1)), a list of field elements.

Steps: defined in Section 5.
```

```
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.
```

```
clear_cofactor(Q)

Input: Q, a point on the elliptic curve E.
Output: P, a point in G.
Steps: defined in Section 7.
```

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

For the API designer…

Syntax specification

```
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
```

```
6.7.1.  Elligator 2 method

   Bernstein, Hamburg, Krasnova, and Lange give a mapping that applies
   to any curve with a point of order 2 [BHKL13], which they call
   Elligator 2.

   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.

   Sign of t: this mapping fixes the sign of t as specified in [BHKL13].
   No additional adjustment is required.

   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).
```

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

For the implementer…

Implementation description

```
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
```

Implementation specification should given implementers confidence in their implementation

F.3. Elligator 2 method

This section gives a straight-line implementation of the Elligator 2 method for any Montgomery curve of the form given in Section 6.7. See Section 6.7.1 for information on the constants used in this mapping.

Appendix G.2 gives optimized straight-line procedures that apply to specific classes of curves and base fields, including curve25519 and curve448 [RFC7748].

```
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
```

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
```

1. Is the specification easy to understand and use?

2. Will the specification yield consistent and correct implementations?

A way forward

Remember the Audience

Align presentation styles

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

Consistency is Key

Reduce cognitive load

Improve clarity by reusing concepts and notation

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

CFRG strives to produce high quality specifications of cryptographic objects

… but consistency across drafts is lacking

Embrace Formality

Adopt formally verified toolchains

Minimize inconsistencies between functional and implementation descriptions

- Can formally verified reference implementations (hacspec) offer a way forward?
- What is the simplest, most approachable reference implementation format? (Python, C/C++, Sage, Rust, etc)

… Unclear if these existing languages with formally verified toolchains are ready for production use in long-lived specifications

Wrapping up

Summary

CFRG specifications (... **standards** …) are important to the community

More work to be done to improve consistency across drafts

- Share terminology, concepts, and notation? Share reference implementations?
- Reference implementation requirements and reviews? Common requirements for syntax (APIs)?

Explore applicability of formal methods for specification