Workgroup: CFRG
Internet-Draft: draft-irtf-cfrg-frost-09
Published: 27 September 2022
Intended Status: Informational
Expires: 31 March 2023
Authors: D. Connolly
Zcash Foundation
C. Komlo
University of Waterloo, Zcash Foundation
I. Goldberg
University of Waterloo
C. A. Wood
Cloudflare

Two-Round Threshold Schnorr Signatures with FROST

Abstract

In this draft, we present the two-round signing variant of FROST, a Flexible Round-Optimized Schnorr Threshold signature scheme. FROST signatures can be issued after a threshold number of entities cooperate to issue a signature, allowing for improved distribution of trust and redundancy with respect to a secret key. Further, this draft specifies signatures that are compatible with [RFC8032]. However, unlike [RFC8032], the protocol for producing signatures in this draft is not deterministic, so as to ensure protection against a key-recovery attack that is possible when even only one signer participant is malicious.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the Crypto Forum Research Group mailing list (cfrg@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=cfrg.

Source for this draft and an issue tracker can be found at https://github.com/cfrg/draft-irtf-cfrg-frost.

Status of This Memo

at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 31 March 2023.

Table of Contents

1. Introduction

DISCLAIMER: This is a work-in-progress draft of FROST.

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH The source for this draft is maintained in GitHub. Suggested changes should be submitted as pull requests at https://github.com/cfrg/draft-irtf-cfrg-frost. Instructions are on that page as well.

Unlike signatures in a single-party setting, threshold signatures require cooperation among a threshold number of signing participants each holding a share of a common private key. The security of threshold schemes in general assumes that an adversary can corrupt strictly fewer than a threshold number of signer participants.

This document presents a variant of a Flexible Round-Optimized Schnorr Threshold (FROST) signature scheme originally defined in [FROST20]. FROST reduces network overhead during threshold signing operations while employing a novel technique to protect against forgery attacks applicable to prior Schnorr-based threshold signature constructions. The variant of FROST presented in this document requires two rounds to compute a signature. Single-round signing with FROST is out of scope.

For select ciphersuites, the signatures produced by this draft are compatible with [RFC8032]. However, unlike [RFC8032], signatures produced by FROST are not deterministic, since deriving nonces deterministically allows for a complete key-recovery attack in multi-party discrete logarithm-based signatures, such as FROST.

While an optimization to FROST was shown in [Schnorr21] that reduces scalar multiplications from linear in the number of signing participants to constant, this draft does not specify that optimization due to the malleability that this optimization introduces, as shown in [StrongerSec22]. Specifically, this optimization removes the guarantee that the set of signer participants that started round one of the protocol is the same set of signing participants that produced the signature output by round two.

Key generation for FROST signing is out of scope for this document. However, for completeness, key generation with a trusted dealer is specified in Appendix C.

1.1. Change Log

draft-09

- *Add single-signer signature generation to complement RFC8032 functions (#293)
- *Address Thomas Pornin review comments from https://mailarchive.ietf.org/arch/msg/crypto-panel/bPyYzwtHlCj00g8YF1tjj-iYP2c/ (#292, #291, #290, #289, #287, #286, #285, #282, #281, #280, #279, #278, #277, #276, #275, #273, #272, #267)
- *Correct Ed448 ciphersuite (#246)
- *Various editorial changes (#241, #240)

draft-08

- *Add notation for Scalar multiplication (#237)
- *Add secp2561k1 ciphersuite (#223)
- *Remove RandomScalar implementation details (#231)
- *Add domain separation for message and commitment digests (#228)

draft-07

- *Fix bug in per-rho signer computation (#222)

draft-06

- Make verification a per-ciphersuite functionality (#219)
- Use per-signer values of rho to mitigate protocol malleability (#217)
- Correct prime-order subgroup checks (#215, #211)
- Fix bug in ed25519 ciphersuite description (#205)
- Various editorial improvements (#208, #209, #210, #218)

draft-05

- Update test vectors to include version string (#202, #203)
- Rename THRESHOLD_LIMIT to MIN_PARTICIPANTS (#192)
- Use non-contiguous signers for the test vectors (#187)
- Add more reasoning why the coordinator MUST abort (#183)
- Add a function to generate nonces (#182)
- Add MUST that all participants have the same view of VSS commitment (#174)
- Use THRESHOLD_LIMIT instead of t and MAX_PARTICIPANTS instead of n (#171)
- Specify what the dealer is trusted to do (#166)
- Clarify types of NUM_PARTICIPANTS and THRESHOLD_LIMIT (#165)
- Assert that the network channel used for signing should be authenticated (#163)
- Remove wire format section (#156)
- Update group commitment derivation to have a single scalarmul (#150)
- Use RandomNonzeroScalar for single-party Schnorr example (#148)
- Fix group notation and clarify member functions (#145)
- Update existing implementations table (#136)

* Various editorial improvements (#135, #143, #147, #149, #153, #158, #162, #167, #168, #169, #170, #175, #176, #177, #178, #184, #186, #193, #198, #199)

draft-04

* Added methods to verify VSS commitments and derive group info (#126, #132).

* Changed check for participants to consider only nonnegative numbers (#133).

* Changed sampling for secrets and coefficients to allow the zero element (#130).

* Split test vectors into separate files (#129)

* Update wire structs to remove commitment shares where not necessary (#128)

* Add failure checks (#127)

* Update group info to include each participant's key and clarify how public key material is obtained (#120, #121).

* Define cofactor checks for verification (#118)

* Various editorial improvements and add contributors (#124, #123, #119, #116, #113, #109)

draft-03

* Refactor the second round to use state from the first round (#94).

* Ensure that verification of signature shares from the second round uses commitments from the first round (#94).

* Clarify RFC8032 interoperability based on PureEdDSA (#86).

* Specify signature serialization based on element and scalar serialization (#85).

* Fix hash function domain separation formatting (#83).

* Make trusted dealer key generation deterministic (#104).

* Add additional constraints on participant indexes and nonce usage (#105, #103, #98, #97).

*Apply various editorial improvements.

draft-02

*Fully specify both rounds of FROST, as well as trusted dealer key generation.

*Add ciphersuites and corresponding test vectors, including suites for RFC8032 compatibility.

*Refactor document for editorial clarity.

draft-01

*Specify operations, notation and cryptographic dependencies.

draft-00

*Outline CFRG draft based on draft-komlo-frost.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The following notation is used throughout the document.

*random_bytes(n): Outputs n bytes, sampled uniformly at random using a cryptographically secure pseudorandom number generator (CSPRNG).

*count(i, L): Outputs the number of times the element i is represented in the list L.

*len(l): Outputs the length of input list l, e.g., len([1,2,3]) = 3).

*reverse(l): Outputs the list l in reverse order, e.g., reverse([1,2,3]) = [3,2,1].

*range(a, b): Outputs a list of integers from a to b-1 in ascending order, e.g., range(1, 4) = [1,2,3].

*pow(a, b): Outputs the integer result of a to the power of b, e.g., pow(2, 3) = 8.

*|| denotes concatenation of byte strings, i.e., x || y denotes the byte string x, immediately followed by the byte string y, with no extra separator, yielding xy.

*nil denotes an empty byte string.

Unless otherwise stated, we assume that secrets are sampled uniformly at random using a cryptographically secure pseudorandom number generator (CSPRNG); see [RFC4086] for additional guidance on the generation of random numbers.

3. Cryptographic Dependencies

FROST signing depends on the following cryptographic constructs:

*Prime-order Group, Section 3.1;

*Cryptographic hash function, Section 3.2;

These are described in the following sections.

3.1. Prime-Order Group

FROST depends on an abelian group of prime order p. We represent this group as the object G that additionally defines helper functions described below. The group operation for G is addition + with identity element I. For any elements A and B of the group G, A + B = B + A is also a member of G. Also, for any A in G, there exists an element -A such that A + (-A) = (-A) + A = I. For convenience, we use - to denote subtraction, e.g., A - B = A + (-B). Integers, taken modulo the group order p, are called scalars; arithmetic operations on scalars are implicitly performed modulo p. Since p is prime, scalars form a finite field. Scalar multiplication is equivalent to the repeated application of the group operation on an element A with itself r-1 times, denoted as ScalarMult(A, r). We denote the sum, difference, and product of two scalars using the +, -, and * operators, respectively. (Note that this means + may refer to group element addition or scalar addition, depending on types of the operands.) For any element A, ScalarMult(A, p) = I. We denote B as a fixed generator of the group. Scalar base multiplication is equivalent to the repeated application of the group operation B with itself r-1 times, this is denoted as ScalarBaseMult(r). The set of scalars corresponds to GF(p), which we refer to as the scalar field. This document uses types Element and Scalar to denote elements of the group G and its set of scalars, respectively. We denote Scalar(x) as the conversion of integer input x to the corresponding Scalar value with the same numeric value. For example, Scalar(1) yields a Scalar representing the value 1. We denote equality comparison as == and assignment of values by =. Finally, it is assumed that group element addition, negation, and equality

comparisons can be efficiently computed for arbitrary group elements.

We now detail a number of member functions that can be invoked on G.

- *Order(): Outputs the order of G (i.e. p).

- *Identity(): Outputs the identity Element of the group (i.e. I).

- *RandomScalar(): Outputs a random Scalar element in GF(p), i.e., a random scalar in [0, p - 1].

- *ScalarMult(A, k): Output the scalar multiplication between Element A and Scalar k.

- *ScalarBaseMult(k): Output the scalar multiplication between Scalar k and the group generator B.

- *SerializeElement(A): Maps an Element A to a canonical byte array buf of fixed length Ne.

- *DeserializeElement(buf): Attempts to map a byte array buf to an Element A, and fails if the input is not the valid canonical byte representation of an element of the group. This function can raise a DeserializeError if deserialization fails or A is the identity element of the group; see Section 6 for group-specific input validation steps.

- *SerializeScalar(s): Maps a Scalar s to a canonical byte array buf of fixed length Ns.

- *DeserializeScalar(buf): Attempts to map a byte array buf to a Scalar s. This function can raise a DeserializeError if deserialization fails; see Section 6 for group-specific input validation steps.

3.2. Cryptographic Hash Function

FROST requires the use of a cryptographically secure hash function, generically written as H, which functions effectively as a random oracle. For concrete recommendations on hash functions which SHOULD be used in practice, see Section 6. Using H, we introduce separate domain-separated hashes, H1, H2, H3, H4, and H5:

- *H1, H2, and H3 map arbitrary byte strings to Scalar elements of the prime-order group scalar field.

- *H4 and H5 are aliases for H with distinct domain separators.

The details of H1, H2, H3, H4, and H5 vary based on ciphersuite. See Section 6 for more details about each.

4. Helper Functions

Beyond the core dependencies, the protocol in this document depends on the following helper operations:

- *Nonce generation, Section 4.1;
- *Polynomial operations, Section 4.2;
- *Encoding operations, Section 4.3;
- *Signature binding Section 4.4, group commitment Section 4.5, and challenge computation Section 4.6.

These sections describes these operations in more detail.

4.1. Nonce generation

To hedge against a bad RNG that outputs predictable values, nonces are generated with the nonce_generate function by combining fresh randomness and with the secret key as input to a domain-separated hash function built from the ciphersuite hash function H. This domain-separated hash function is denoted H3. This function always samples 32 bytes of fresh randomness to ensure that the probability of nonce reuse is at most 2^{-128} as long as no more than 2^{64} signatures are computed by a given signing participant.

```
nonce_generate(secret):

Inputs:
- secret, a Scalar

Outputs: nonce, a Scalar

def nonce_generate(secret):
  random_bytes = random_bytes(32)
  secret_enc = G.SerializeScalar(secret)
  return H3(random_bytes || secret_enc)
```

4.2. Polynomial Operations

This section describes operations on and associated with polynomials over Scalars that are used in the main signing protocol. A polynomial of maximum degree t+1 is represented as a list of t coefficients, where the constant term of the polynomial is in the first position and the highest-degree coefficient is in the last position. A point on the polynomial is a tuple (x, y), where y =

f(x). For notational convenience, we refer to the x-coordinate and y-coordinate of a point p as p.x and p.y, respectively.

4.2.1. Evaluation of a polynomial

This section describes a method for evaluating a polynomial f at a particular input x, i.e., y = f(x) using Horner's method.

```
polynomial_evaluate(x, coeffs):

Inputs:
- x, input at which to evaluate the polynomial, a Scalar
- coeffs, the polynomial coefficients, a list of Scalars

Outputs: Scalar result of the polynomial evaluated at input x

def polynomial_evaluate(x, coeffs):
  value = 0
  for coeff in reverse(coeffs):
    value *= x
    value += coeff
  return value
```

4.2.2. Lagrange coefficients

The function derive_lagrange_coefficient derives a Lagrange coefficient to later perform polynomial interpolation, and is provided a list of x-coordinates as input. Note that derive_lagrange_coefficient does not permit any x-coordinate to equal 0. Lagrange coefficients are used in FROST to evaluate a polynomial f at x-coordinate 0, i.e., f(0), given a list of t other x-coordinates.

```
derive_lagrange_coefficient(x_i, L):

Inputs:
- x_i, an x-coordinate contained in L, a Scalar
- L, the set of x-coordinates, each a Scalar

Outputs: L_i, the i-th Lagrange coefficient

Errors:
- "invalid parameters", if 1) any x-coordinate is equal to 0, 2) if x_
  is not in L, or if 3) any x-coordinate is represented more than once

def derive_lagrange_coefficient(x_i, L):
  if x_i == 0:
    raise "invalid parameters"
  for x_j in L:
    if x_j == 0:
      raise "invalid parameters"
  if x_i not in L:
    raise "invalid parameters"
  for x_j in L:
    if count(x_i, L) > 1:
      raise "invalid parameters"

  numerator = Scalar(1)
  denominator = Scalar(1)
  for x_j in L:
    if x_j == x_i: continue
    numerator *= x_j
    denominator *= x_j - x_i

  L_i = numerator / denominator
  return L_i
```

4.3. List Operations

This section describes helper functions that work on lists of values produced during the FROST protocol. The following function encodes a list of participant commitments into a bytestring for use in the FROST protocol.

```
  Inputs:
  - commitment_list = [(i, hiding_nonce_commitment_i, binding_nonce_comm
    a list of commitments issued by each participant, where each element
    indicates the participant identifier i and their two commitment Elem
    (hiding_nonce_commitment_i, binding_nonce_commitment_i). This list M
    in ascending order by participant identifier.

  Outputs: A byte string containing the serialized representation of com

  def encode_group_commitment_list(commitment_list):
    encoded_group_commitment = nil
    for (identifier, hiding_nonce_commitment, binding_nonce_commitment)
      encoded_commitment = G.SerializeScalar(identifier) ||
                           G.SerializeElement(hiding_nonce_commitment) |
                           G.SerializeElement(binding_nonce_commitment)
      encoded_group_commitment = encoded_group_commitment || encoded_com
    return encoded_group_commitment
```

The following function is used to extract participant identifiers from a commitment list.

```
  Inputs:
  - commitment_list = [(i, hiding_nonce_commitment_i, binding_nonce_comm
    a list of commitments issued by each participant, where each element
    indicates the participant identifier i and their two commitment Elem
    (hiding_nonce_commitment_i, binding_nonce_commitment_i). This list M
    in ascending order by participant identifier.

  Outputs: A list of participant identifiers

def participants_from_commitment_list(commitment_list):
  identifiers = []
  for (identifier, _, _) in commitment_list:
    identifiers.append(identifier)
  return identifiers
```

The following function is used to extract a binding factor from a list of binding factors.

```
  Inputs:
  - binding_factor_list = [(i, binding_factor), ...],
    a list of binding factors for each participant, where each element i
    indicates the participant identifier i and their binding factor. Thi
    in ascending order by participant identifier.
  - identifier, participant identifier, a Scalar.

  Outputs: A Scalar value.

  Errors: "invalid participant", when the designated participant is not

def binding_factor_for_participant(binding_factor_list, identifier):
  for (i, binding_factor) in binding_factor_list:
    if identifier == i:
      return binding_factor
  raise "invalid participant"
```

4.4. Binding Factors Computation

This section describes the subroutine for computing binding factors based on the participant commitment list and message to be signed.

```
  Inputs:
  - commitment_list = [(i, hiding_nonce_commitment_i, binding_nonce_comm
    a list of commitments issued by each participant, where each element
    indicates the participant identifier i and their two commitment Elem
    (hiding_nonce_commitment_i, binding_nonce_commitment_i). This list M
    in ascending order by participant identifier.
  - msg, the message to be signed.

  Outputs: A list of (identifier, Scalar) tuples representing the bindin

  def compute_binding_factors(commitment_list, msg):
    msg_hash = H4(msg)
    encoded_commitment_hash = H5(encode_group_commitment_list(commitment
    rho_input_prefix = msg_hash || encoded_commitment_hash

    binding_factor_list = []
    for (identifier, hiding_nonce_commitment, binding_nonce_commitment)
      rho_input = rho_input_prefix || G.SerializeScalar(identifier)
      binding_factor = H1(rho_input)
      binding_factor_list.append((identifier, binding_factor))
    return binding_factor_list
```

4.5. Group Commitment Computation

This section describes the subroutine for creating the group commitment from a commitment list.

```
Inputs:
- commitment_list =
   [(i, hiding_nonce_commitment_i, binding_nonce_commitment_i), ...],
  of commitments issued by each participant, where each element in the
  indicates the participant identifier i and their two commitment Elem
  (hiding_nonce_commitment_i, binding_nonce_commitment_i). This list M
  sorted in ascending order by participant identifier.
- binding_factor_list = [(i, binding_factor), ...],
  a list of (identifier, Scalar) tuples representing the binding facto
  for the given identifier. This list MUST be sorted in ascending orde

Outputs: An Element in G representing the group commitment

def compute_group_commitment(commitment_list, binding_factor_list):
  group_commitment = G.Identity()
  for (identifier, hiding_nonce_commitment, binding_nonce_commitment)
    binding_factor = binding_factor_for_participant(binding_factors, i
    group_commitment = group_commitment +
      hiding_nonce_commitment + G.ScalarMult(binding_nonce_commitment,
  return group_commitment
```

4.6. Signature Challenge Computation

This section describes the subroutine for creating the per-message challenge.

```
Inputs:
- group_commitment, an Element in G representing the group commitment
- group_public_key, public key corresponding to the group signing key,
  Element in G.
- msg, the message to be signed.

Outputs: A Scalar representing the challenge

def compute_challenge(group_commitment, group_public_key, msg):
  group_comm_enc = G.SerializeElement(group_commitment)
  group_public_key_enc = G.SerializeElement(group_public_key)
  challenge_input = group_comm_enc || group_public_key_enc || msg
  challenge = H2(challenge_input)
  return challenge
```

5. Two-Round FROST Signing Protocol

This section describes the two-round variant of the FROST threshold signature protocol for producing Schnorr signatures. The protocol is configured to run with a selection of NUM_PARTICIPANTS signer participants and a Coordinator. NUM_PARTICIPANTS is a positive integer at least MIN_PARTICIPANTS but no larger than MAX_PARTICIPANTS, where MIN_PARTICIPANTS < MAX_PARTICIPANTS, MIN_PARTICIPANTS is a positive integer and MAX_PARTICIPANTS is a

positive integer less than the group order. A signer participant, or simply participant, is an entity that is trusted to hold and use a signing key share. The Coordinator is an entity with the following responsibilities:

1. Determining which participants will participate (at least MIN_PARTICIPANTS in number);

2. Coordinating rounds (receiving and forwarding inputs among participants); and

3. Aggregating signature shares output by each participant, and publishing the resulting signature.

FROST assumes that all participants, including the Coordinator and the set of participants, are chosen externally to the protocol. Note that it is possible to deploy the protocol without a distinguished Coordinator; see Section 7.3 for more information.

FROST produces signatures that are indistinguishable from those produced with a single participant using a signing key s with corresponding public key PK, where s is a Scalar value and PK = G.ScalarMultBase(s). As a threshold signing protocol, the group signing key s is secret-shared amongst each participant and used to produce signatures. In particular, FROST assumes each participant is configured with the following information:

- An identifier, which is a Scalar value denoted i in the range [1, MAX_PARTICIPANTS] and MUST be distinct from the identifier of every other participant.

- A signing key share sk_i, which is a Scalar value representing the i-th secret share of the group signing key s. The public key corresponding to this signing key share is PK_i = G.ScalarMultBase(sk_i).

Each participant, including the Coordinator, is additionally configured with common group information, denoted "group info," which consists of the following information:

- Group public key, which is an Element in G denoted PK.

- Public keys PK_i for each signer, which are Element values in G denoted PK_i for each i in [1, MAX_PARTICIPANTS].

This document does not specify how this information, including the signing key shares, are configured and distributed to participants. In general, two possible configuration mechanisms are possible: one that requires a single, trusted dealer, and the other which requires performing a distributed key generation protocol. We highlight key

generation mechanism by a trusted dealer in Appendix C for reference.

The signing variant of FROST in this document requires participants to perform two network rounds: 1) generating and publishing commitments, and 2) signature share generation and publication. The first round serves for each participant to issue a commitment to a nonce. The second round receives commitments for all participants as well as the message, and issues a signature share with respect to that message. The Coordinator performs the coordination of each of these rounds. At the end of the second round, the Coordinator then performs an aggregation step and outputs the final signature. This complete interaction is shown in Figure 1.

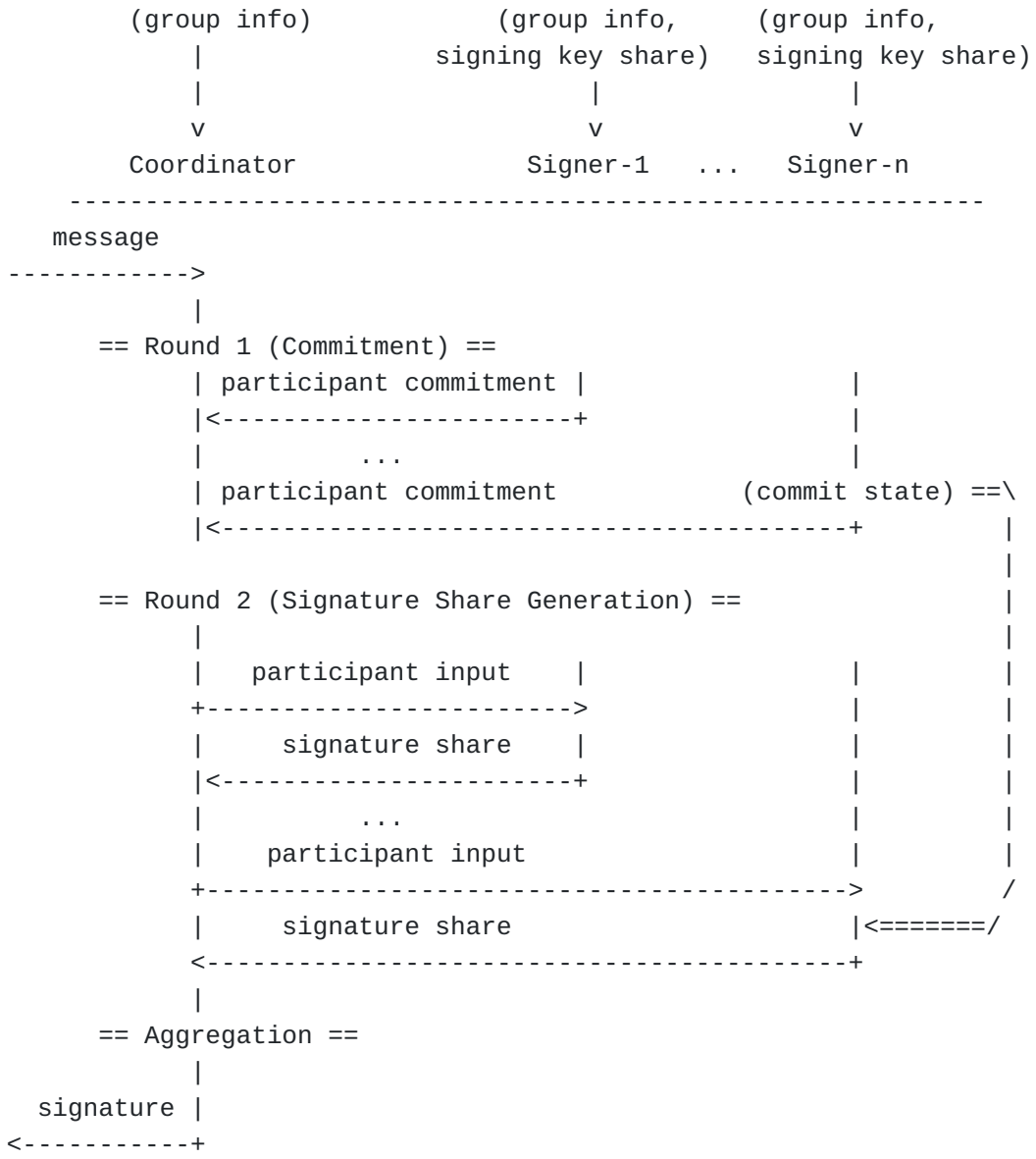

Figure 1: FROST signature overview

Details for round one are described in Section 5.1, and details for round two are described in Section 5.2. Note that each participant persists some state between both rounds, and this state is deleted as described in Section 5.2. The final Aggregation step is described in Section 5.3.

FROST assumes that all inputs to each round, especially those of which are received over the network, are validated before use. In particular, this means that any value of type Element or Scalar is deserialized using DeserializeElement and DeserializeScalar, respectively, as these functions perform the necessary input validation steps.

FROST assumes reliable message delivery between the Coordinator and participants in order for the protocol to complete. An attacker masquerading as another participant will result only in an invalid signature; see Section 7. However, in order to identify any participant which has misbehaved (resulting in the protocol aborting) to take actions such as excluding them from future signing operations, we assume that the network channel is additionally authenticated; confidentiality is not required.

5.1. Round One - Commitment

Round one involves each participant generating nonces and their corresponding public commitments. A nonce is a pair of Scalar values, and a commitment is a pair of Element values. Each participant's behavior in this round is described by the commit function below. Note that this function invokes nonce_generate twice, once for each type of nonce produced. The output of this function is a pair of secret nonces (hiding_nonce, binding_nonce) and their corresponding public commitments (hiding_nonce_commitment, binding_nonce_commitment).

Inputs: sk_i, the secret key share, a Scalar

Outputs: (nonce, comm), a tuple of nonce and nonce commitment pairs, where each value in the nonce pair is a Scalar and each value in the nonce commitment pair is an Element

```
def commit(sk_i):
  hiding_nonce = nonce_generate(sk_i)
  binding_nonce = nonce_generate(sk_i)
  hiding_nonce_commitment = G.ScalarBaseMult(hiding_nonce)
  binding_nonce_commitment = G.ScalarBaseMult(binding_nonce)
  nonce = (hiding_nonce, binding_nonce)
  comm = (hiding_nonce_commitment, binding_nonce_commitment)
  return (nonce, comm)
```

The outputs nonce and comm from participant P_i should both be stored locally and kept for use in the second round. The nonce value is secret and MUST NOT be shared, whereas the public output comm is sent to the Coordinator. The nonce values produced by this function MUST NOT be reused in more than one invocation of FROST, and it MUST be generated from a source of secure randomness.

5.2. Round Two - Signature Share Generation

In round two, the Coordinator is responsible for sending the message to be signed, and for choosing which participants will participate (of number at least MIN_PARTICIPANTS). Signers additionally require locally held data; specifically, their private key and the nonces corresponding to their commitment issued in round one.

The Coordinator begins by sending each participant the message to be signed along with the set of signing commitments for all participants in the participant list. Each participant MUST validate the inputs before processing the Coordinator's request. In particular, the Signer MUST validate commitment_list, deserializing each group Element in the list using DeserializeElement from Section 3.1. If deserialization fails, the Signer MUST abort the protocol. Moreover, each participant MUST ensure that their identifier as well as their commitment as from the first round appears in commitment_list. Applications which require that participants not process arbitrary input messages are also required to also perform relevant application-layer input validation checks; see Section 7.5 for more details.

Upon receipt and successful input validation, each Signer then runs the following procedure to produce its own signature share.

```
Inputs:
- identifier, Identifier i of the participant. Note identifier will ne
- sk_i, Signer secret key share, a Scalar.
- group_public_key, public key corresponding to the group signing key,
  an Element in G.
- nonce_i, pair of Scalar values (hiding_nonce, binding_nonce) generat
  round one.
- msg, the message to be signed (sent by the Coordinator).
- commitment_list =
    [(j, hiding_nonce_commitment_j, binding_nonce_commitment_j), ...],
  list of commitments issued in Round 1 by each participant and sent b
  Each element in the list indicates the participant identifier j and
  Element values (hiding_nonce_commitment_j, binding_nonce_commitment_
  This list MUST be sorted in ascending order by participant identifie

Outputs: a Scalar value representing the signature share

def sign(identifier, sk_i, group_public_key, nonce_i, msg, commitment_
  # Compute the binding factor(s)
  binding_factor_list = compute_binding_factors(commitment_list, msg)
  binding_factor = binding_factor_for_participant(binding_factor_list,

  # Compute the group commitment
  group_commitment = compute_group_commitment(commitment_list, binding

  # Compute Lagrange coefficient
  participant_list = participants_from_commitment_list(commitment_list
  lambda_i = derive_lagrange_coefficient(identifier, participant_list)

  # Compute the per-message challenge
  challenge = compute_challenge(group_commitment, group_public_key, ms

  # Compute the signature share
  (hiding_nonce, binding_nonce) = nonce_i
  sig_share = hiding_nonce + (binding_nonce * binding_factor) + (lambd

  return sig_share

 The output of this procedure is a signature share. Each participant
 then sends these shares back to the Coordinator. Each participant
 MUST delete the nonce and corresponding commitment after this round
 completes, and MUST use the nonce to generate at most one signature
 share.

 Note that the lambda_i value derived during this procedure does not
 change across FROST signing operations for the same signing group.
 As such, participants can compute it once and store it for reuse
 across signing sessions.
```

Upon receipt from each Signer, the Coordinator MUST validate the input signature share using DeserializeElement. If validation fails, the Coordinator MUST abort the protocol. If validation succeeds, the Coordinator then verifies the set of signature shares using the following procedure.

5.3. Signature Share Verification and Aggregation

After participants perform round two and send their signature shares to the Coordinator, the Coordinator verifies each signature share for correctness. In particular, for each participant, the Coordinator uses commitment pairs generated during round one and the signature share generated during round two, along with other group parameters, to check that the signature share is valid using the following procedure.

```
Inputs:
- identifier, Identifier i of the participant. Note: identifier MUST n
- PK_i, the public key for the ith participant, where PK_i = G.ScalarB
  an Element in G
- comm_i, pair of Element values in G (hiding_nonce_commitment, bindin
  generated in round one from the ith participant.
- sig_share_i, a Scalar value indicating the signature share as produc
  round two from the ith participant.
- commitment_list =
    [(j, hiding_nonce_commitment_j, binding_nonce_commitment_j), ...],
  list of commitments issued in Round 1 by each participant, where eac
  in the list indicates the participant identifier j and their two com
  Element values (hiding_nonce_commitment_j, binding_nonce_commitment_
  This list MUST be sorted in ascending order by participant identifie
- group_public_key, public key corresponding to the group signing key,
  an Element in G.
- msg, the message to be signed.

Outputs: True if the signature share is valid, and False otherwise.

def verify_signature_share(identifier, PK_i, comm_i, sig_share_i, comm
                           group_public_key, msg):
  # Compute the binding factors
  binding_factor_list = compute_binding_factors(commitment_list, msg)
  binding_factor = binding_factor_for_participant(binding_factor_list,

  # Compute the group commitment
  group_commitment = compute_group_commitment(commitment_list, binding

  # Compute the commitment share
  (hiding_nonce_commitment, binding_nonce_commitment) = comm_i
  comm_share = hiding_nonce_commitment + G.ScalarMult(binding_nonce_co

  # Compute the challenge
  challenge = compute_challenge(group_commitment, group_public_key, ms

  # Compute Lagrange coefficient
  participant_list = participants_from_commitment_list(commitment_list
  lambda_i = derive_lagrange_coefficient(identifier, participant_list)

  # Compute relation values
  l = G.ScalarBaseMult(sig_share_i)
  r = comm_share + G.ScalarMult(PK_i, challenge * lambda_i)

  return l == r

 If any signature share fails to verify, i.e., if
 verify_signature_share returns False for any participant share, the
 Coordinator MUST abort the protocol for correctness reasons (this is
 true regardless of the size or makeup of the signing set selected by
```

the Coordinator). Excluding one participant means that their nonce will not be included in the joint response z and consequently the output signature will not verify. This is because the group commitment will be with respect to a different signing set than the the aggregated response.

Otherwise, if all shares from participants that participated in Rounds 1 and 2 are valid, the Coordinator performs the aggregate operation and publishes the resulting signature.

```
Inputs:
- group_commitment, the group commitment returned by compute_group_com
  an Element in G.
- sig_shares, a set of signature shares z_i, Scalar values, for each p
  of length NUM_PARTICIPANTS, where MIN_PARTICIPANTS <= NUM_PARTICIPAN

Outputs: (R, z), a Schnorr signature consisting of an Element R and Sc

def aggregate(group_commitment, sig_shares):
  z = 0
  for z_i in sig_shares:
    z = z + z_i
  return (group_commitment, z)
```

The output signature (R, z) from the aggregation step MUST be encoded as follows (using notation from Section 3 of [TLS]):

```
struct {
  opaque R_encoded[Ne];
  opaque z_encoded[Ns];
} Signature;
```

Where Signature.R_encoded is G.SerializeElement(R) and Signature.z_encoded is G.SerializeScalar(z).

6. Ciphersuites

A FROST ciphersuite must specify the underlying prime-order group details and cryptographic hash function. Each ciphersuite is denoted as (Group, Hash), e.g., (ristretto255, SHA-512). This section contains some ciphersuites. Each ciphersuite also includes a context string, denoted contextString, which is an ASCII string literal (with no NULL terminating character).

The RECOMMENDED ciphersuite is (ristretto255, SHA-512) Section 6.2. The (Ed25519, SHA-512) ciphersuite is included for backwards compatibility with [RFC8032].

The DeserializeElement and DeserializeScalar functions instantiated for a particular prime-order group corresponding to a ciphersuite

MUST adhere to the description in Section 3.1. Validation steps for these functions are described for each the ciphersuites below. Future ciphersuites MUST describe how input validation is done for DeserializeElement and DeserializeScalar.

Each ciphersuite includes explicit instructions for verifying signatures produced by FROST. Note that these instructions are equivalent to those produced by a single participant.

6.1. FROST(Ed25519, SHA-512)

This ciphersuite uses edwards25519 for the Group and SHA-512 for the Hash function H meant to produce signatures indistinguishable from Ed25519 as specified in [RFC8032]. The value of the contextString parameter is "FROST-ED25519-SHA512-v8".

*Group: edwards25519 [RFC8032]

-Order(): Return 2^252 + 27742317777372353535851937790883648493 (see [RFC7748])

-Identity(): As defined in [RFC7748].

-RandomScalar(): Implemented by returning a uniformly random Scalar in the range [0, G.Order() - 1]. Refer to Appendix D for implementation guidance.

-SerializeElement(A): Implemented as specified in [RFC8032], Section 5.1.2.

-DeserializeElement(buf): Implemented as specified in [RFC8032], Section 5.1.3. Additionally, this function validates that the resulting element is not the group identity element and is in the prime-order subgroup. The latter check can be implemented by multiplying the resulting point by the order of the group and checking that the result is the identity element.

-SerializeScalar(s): Implemented by outputting the little-endian 32-byte encoding of the Scalar value with the top three bits set to zero.

-DeserializeScalar(buf): Implemented by attempting to deserialize a Scalar from a little-endian 32-byte string. This function can fail if the input does not represent a Scalar in the range [0, G.Order() - 1]. Note that this means the top three bits of the input MUST be zero.

*Hash (H): SHA-512

  -H1(m): Implemented by computing H(contextString || "rho" || m), interpreting the 64-byte digest as a little-endian integer, and reducing the resulting integer modulo 2^252+27742317777372353535851937790883648493.

  -H2(m): Implemented by computing H(m), interpreting the 64-byte digest as a little-endian integer, and reducing the resulting integer modulo 2^252+27742317777372353535851937790883648493.

  -H3(m): Implemented by computing H(contextString || "nonce" || m), interpreting the 64-byte digest as a little-endian integer, and reducing the resulting integer modulo 2^252+27742317777372353535851937790883648493.

  -H4(m): Implemented by computing H(contextString || "msg" || m).

  -H5(m): Implemented by computing H(contextString || "com" || m).

Normally H2 would also include a domain separator, but for backwards compatibility with [RFC8032], it is omitted.

Signature verification is as specified in Section 5.1.7 of [RFC8032] with the constraint that implementations MUST check the group equation [8][S]B = [8]R + [8][k]A'. The alternative check [S]B = R + [k]A' is not safe or interoperable in practice. Note that optimizations for this check exist; see [Pornin22].

6.2. FROST(ristretto255, SHA-512)

This ciphersuite uses ristretto255 for the Group and SHA-512 for the Hash function H. The value of the contextString parameter is "FROST-RISTRETTO255-SHA512-v8".

*Group: ristretto255 [RISTRETTO]

  -Order(): Return 2^252 + 27742317777372353535851937790883648493 (see [RISTRETTO])

  -Identity(): As defined in [RISTRETTO].

  -RandomScalar(): Implemented by returning a uniformly random Scalar in the range [0, G.Order() - 1]. Refer to Appendix D for implementation guidance.

  -SerializeElement(A): Implemented using the 'Encode' function from [RISTRETTO].

- DeserializeElement(buf): Implemented using the 'Decode' function from [RISTRETTO]. Additionally, this function validates that the resulting element is not the group identity element.

- SerializeScalar(s): Implemented by outputting the little-endian 32-byte encoding of the Scalar value with the top three bits set to zero.

- DeserializeScalar(buf): Implemented by attempting to deserialize a Scalar from a little-endian 32-byte string. This function can fail if the input does not represent a Scalar in the range [0, G.Order() - 1]. Note that this means the top three bits of the input MUST be zero.

* Hash (H): SHA-512

  - H1(m): Implemented by computing H(contextString || "rho" || m) and mapping the output to a Scalar as described in [RISTRETTO], Section 4.4.

  - H2(m): Implemented by computing H(contextString || "chal" || m) and mapping the output to a Scalar as described in [RISTRETTO], Section 4.4.

  - H3(m): Implemented by computing H(contextString || "nonce" || m) and mapping the output to a Scalar as described in [RISTRETTO], Section 4.4.

  - H4(m): Implemented by computing H(contextString || "msg" || m).

  - H5(m): Implemented by computing H(contextString || "com" || m).

Signature verification is as specified in Appendix B.

6.3. FROST(Ed448, SHAKE256)

This ciphersuite uses edwards448 for the Group and SHAKE256 for the Hash function H meant to produce signatures indistinguishable from Ed448 as specified in [RFC8032]. The value of the contextString parameter is "FROST-ED448-SHAKE256-v8".

* Group: edwards448 [RFC8032]

  - Order(): Return 2^446 - 138180668098951153520073867485154268803366924748821786098945475038 85

- -Identity(): As defined in [RFC7748].

- -RandomScalar(): Implemented by returning a uniformly random Scalar in the range [0, G.Order() - 1]. Refer to Appendix D for implementation guidance.

- -SerializeElement(A): Implemented as specified in [RFC8032], Section 5.2.2.

- -DeserializeElement(buf): Implemented as specified in [RFC8032], Section 5.2.3. Additionally, this function validates that the resulting element is not the group identity element.

- -SerializeScalar(s): Implemented by outputting the little-endian 48-byte encoding of the Scalar value.

- -DeserializeScalar(buf): Implemented by attempting to deserialize a Scalar from a little-endian 48-byte string. This function can fail if the input does not represent a Scalar in the range [0, G.Order() - 1].

* Hash (H): SHAKE256

- -H1(m): Implemented by computing H(contextString || "rho" || m), interpreting the 114-byte digest as a little-endian integer, and reducing the resulting integer modulo 2^446 - 138180668098951153520073867485154268803366924748821786098945475038 85.

- -H2(m): Implemented by computing H("SigEd448" || 0 || 0 || m), interpreting the 114-byte digest as a little-endian integer, and reducing the resulting integer modulo 2^446 - 138180668098951153520073867485154268803366924748821786098945475038 85.

- -H3(m): Implemented by computing H(contextString || "nonce" || m), interpreting the 114-byte digest as a little-endian integer, and reducing the resulting integer modulo 2^446 - 138180668098951153520073867485154268803366924748821786098945475038 85.

- -H4(m): Implemented by computing H(contextString || "msg" || m).

- -H5(m): Implemented by computing H(contextString || "com" || m).

Normally H2 would also include a domain separator, but for backwards compatibility with [RFC8032], it is omitted.

Signature verification is as specified in Section 5.2.7 of [RFC8032] with the constraint that implementations MUST check the group equation [4][S]B = [4]R + [4][k]A'. The alternative check [S]B = R + [k]A' is not safe or interoperable in practice. Note that optimizations for this check exist; see [Pornin22].

6.4. FROST(P-256, SHA-256)

This ciphersuite uses P-256 for the Group and SHA-256 for the Hash function H. The value of the contextString parameter is "FROST-P256-SHA256-v8".

*Group: P-256 (secp256r1) [x9.62]

-Order(): Return 0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551

-Identity(): As defined in [x9.62].

-RandomScalar(): Implemented by returning a uniformly random Scalar in the range [0, G.Order() - 1]. Refer to Appendix D for implementation guidance.

-SerializeElement(A): Implemented using the compressed Elliptic-Curve-Point-to-Octet-String method according to [SEC1], yielding a 33 byte output.

-DeserializeElement(buf): Implemented by attempting to deserialize a 33 byte input string to a public key using the compressed Octet-String-to-Elliptic-Curve-Point method according to [SEC1], and then performs partial public-key validation as defined in section 5.6.2.3.4 of [KEYAGREEMENT]. This includes checking that the coordinates of the resulting point are in the correct range, that the point is on the curve, and that the point is not the point at infinity. Additionally, this function validates that the resulting element is not the group identity element. If these checks fail, deserialization returns an error.

-SerializeScalar(s): Implemented using the Field-Element-to-Octet-String conversion according to [SEC1].

-DeserializeScalar(buf): Implemented by attempting to deserialize a Scalar from a 32-byte string using Octet-String-to-Field-Element from [SEC1]. This function can fail if the input does not represent a Scalar in the range [0, G.Order() - 1].

*Hash (H): SHA-256

- H1(m): Implemented as hash_to_field(m, 1) from [HASH-TO-CURVE], Section 5.2 using expand_message_xmd with SHA-256 with parameters DST = contextString || "rho", F set to the scalar field, p set to G.Order(), m = 1, and L = 48.

- H2(m): Implemented as hash_to_field(m, 1) from [HASH-TO-CURVE], Section 5.2 using expand_message_xmd with SHA-256 with parameters DST = contextString || "chal", F set to the scalar field, p set to G.Order(), m = 1, and L = 48.

- H3(m): Implemented as hash_to_field(m, 1) from [HASH-TO-CURVE], Section 5.2 using expand_message_xmd with SHA-256 with parameters DST = contextString || "nonce", F set to the scalar field, p set to G.Order(), m = 1, and L = 48.

- H4(m): Implemented by computing H(contextString || "msg" || m).

- H5(m): Implemented by computing H(contextString || "com" || m).

Signature verification is as specified in Appendix B.

6.5. FROST(secp256k1, SHA-256)

This ciphersuite uses secp256k1 for the Group and SHA-256 for the Hash function H. The value of the contextString parameter is "FROST-secp256k1-SHA256-v8".

*Group: secp256k1 [SEC2]

- Order(): Return 0xfffffffe00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551

- Identity(): As defined in [SEC2].

- RandomScalar(): Implemented by returning a uniformly random Scalar in the range [0, G.Order() - 1]. Refer to Appendix D for implementation guidance.

- SerializeElement(A): Implemented using the compressed Elliptic-Curve-Point-to-Octet-String method according to [SEC1].

- DeserializeElement(buf): Implemented by attempting to deserialize a public key using the compressed Octet-String-to-Elliptic-Curve-Point method according to [SEC1], and then

performs partial public-key validation as defined in section 3.2.2.1 of [SEC1]. This includes checking that the coordinates of the resulting point are in the correct range, that the point is on the curve, and that the point is not the point at infinity. Additionally, this function validates that the resulting element is not the group identity element. If these checks fail, deserialization returns an error.

- SerializeScalar(s): Implemented using the Field-Element-to-Octet-String conversion according to [SEC1].

- DeserializeScalar(buf): Implemented by attempting to deserialize a Scalar from a 32-byte string using Octet-String-to-Field-Element from [SEC1]. This function can fail if the input does not represent a Scalar in the range [0, G.Order() - 1].

* Hash (H): SHA-256

  - H1(m): Implemented as hash_to_field(m, 1) from [HASH-TO-CURVE], Section 5.2 using expand_message_xmd with SHA-256 with parameters DST = contextString || "rho", F set to the scalar field, p set to G.Order(), m = 1, and L = 48.

  - H2(m): Implemented as hash_to_field(m, 1) from [HASH-TO-CURVE], Section 5.2 using expand_message_xmd with SHA-256 with parameters DST = contextString || "chal", F set to the scalar field, p set to G.Order(), m = 1, and L = 48.

  - H3(m): Implemented as hash_to_field(m, 1) from [HASH-TO-CURVE], Section 5.2 using expand_message_xmd with SHA-256 with parameters DST = contextString || "nonce", F set to the scalar field, p set to G.Order(), m = 1, and L = 48.

  - H4(m): Implemented by computing H(contextString || "msg" || m).

  - H5(m): Implemented by computing H(contextString || "com" || m).

Signature verification is as specified in Appendix B.

7. Security Considerations

A security analysis of FROST exists in [FROST20] and [Schnorr21]. The protocol as specified in this document assumes the following threat model.

* Trusted dealer. The dealer that performs key generation is trusted to follow the protocol, although participants still are

able to verify the consistency of their shares via a VSS (verifiable secret sharing) step; see Appendix C.2.

* Unforgeability assuming at most (MIN_PARTICIPANTS-1) corrupted participants. So long as an adversary corrupts fewer than MIN_PARTICIPANTS participants, the scheme remains secure against Existential Unforgeability Under Chosen Message Attack (EUF-CMA) attacks, as defined in [BonehShoup], Definition 13.2.

* Coordinator. We assume the Coordinator at the time of signing does not perform a denial of service attack. A denial of service would include any action which either prevents the protocol from completing or causing the resulting signature to be invalid. Such actions for the latter include sending inconsistent values to participants, such as messages or the set of individual commitments. Note that the Coordinator is *not* trusted with any private information and communication at the time of signing can be performed over a public but reliable channel.

The protocol as specified in this document does not target the following goals:

* Post quantum security. FROST, like plain Schnorr signatures, requires the hardness of the Discrete Logarithm Problem.

* Robustness. In the case of failure, FROST requires aborting the protocol.

* Downgrade prevention. All participants in the protocol are assumed to agree on what algorithms to use.

* Metadata protection. If protection for metadata is desired, a higher-level communication channel can be used to facilitate key generation and signing.

The rest of this section documents issues particular to implementations or deployments.

7.1. Nonce Reuse Attacks

Section 4.1 describes the procedure that participants use to produce nonces during the first round of singing. The randomness produced in this procedure MUST be sampled uniformly at random. The resulting nonces produced via nonce_generate are indistinguishable from values sampled uniformly at random. This requirement is necessary to avoid replay attacks initiated by other participants, which allow for a complete key-recovery attack. The Coordinator MAY further hedge against nonce reuse attacks by tracking participant nonce commitments used for a given group key, at the cost of additional state.

7.2. Protocol Failures

We do not specify what implementations should do when the protocol fails, other than requiring that the protocol abort. Examples of viable failure include when a verification check returns invalid or if the underlying transport failed to deliver the required messages.

7.3. Removing the Coordinator Role

In some settings, it may be desirable to omit the role of the Coordinator entirely. Doing so does not change the security implications of FROST, but instead simply requires each participant to communicate with all other participants. We loosely describe how to perform FROST signing among participants without this coordinator role. We assume that every participant receives as input from an external source the message to be signed prior to performing the protocol.

Every participant begins by performing commit() as is done in the setting where a Coordinator is used. However, instead of sending the commitment to the Coordinator, every participant instead will publish this commitment to every other participant. Then, in the second round, participants will already have sufficient information to perform signing. They will directly perform sign(). All participants will then publish their signature shares to one another. After having received all signature shares from all other participants, each participant will then perform verify_signature_share and then aggregate directly.

The requirements for the underlying network channel remain the same in the setting where all participants play the role of the Coordinator, in that all messages that are exchanged are public and so the channel simply must be reliable. However, in the setting that a player attempts to split the view of all other players by sending disjoint values to a subset of players, the signing operation will output an invalid signature. To avoid this denial of service, implementations may wish to define a mechanism where messages are authenticated, so that cheating players can be identified and excluded.

7.4. Input Message Hashing

FROST signatures do not pre-hash message inputs. This means that the entire message must be known in advance of invoking the signing protocol. Applications can apply pre-hashing in settings where storing the full message is prohibitively expensive. In such cases, pre-hashing MUST use a collision-resistant hash function with a security level commensurate with the security in inherent to the ciphersuite chosen. It is RECOMMENDED that applications which choose

to apply pre-hashing use the hash function (H) associated with the chosen ciphersuite in a manner similar to how H4 is defined. In particular, a different prefix SHOULD be used to differentiate this pre-hash from H4. One possible example is to construct this pre-hash over message m as H(contextString \|\| "pre-hash" \|\| m).

7.5. Input Message Validation

Some applications may require that participants only process messages of a certain structure. For example, in digital currency applications wherein multiple participants may collectively sign a transaction, it is reasonable to require that each participant check the input message to be a syntactically valid transaction.

As another example, use of threshold signatures in [TLS] to produce signatures of transcript hashes might require the participants receive the source handshake messages themselves, and recompute the transcript hash which is used as input message to the signature generation process, so that they can verify that they are signing a proper TLS transcript hash and not some other data.

In general, input message validation is an application-specific consideration that varies based on the use case and threat model. However, it is RECOMMENDED that applications take additional precautions and validate inputs so that participants do not operate as signing oracles for arbitrary messages.

8. Contributors

9. References

9.1. Normative References

[HASH-TO-CURVE] Faz-Hernández, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-16, 15 June 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-16>.

[KEYAGREEMENT] Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-56ar3, April 2018, <https://doi.org/10.6028/nist.sp.800-56ar3>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/rfc/rfc8032>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RISTRETTO] de Valence, H., Grigg, J., Hamburg, M., Lovecruft, I., Tankersley, G., and F. Valsorda, "The ristretto255 and decaf448 Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-ristretto255-decaf448-03, 25 February 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-03>.

[SEC1] "Elliptic Curve Cryptography, Standards for Efficient Cryptography Group, ver. 2", 2009, <https://secg.org/sec1-v2.pdf>.

[SEC2] "Recommended Elliptic Curve Domain Parameters, Standards for Efficient Cryptography Group, ver. 2", 2010, <https://secg.org/sec2-v2.pdf>.

[x9.62] ANS, "Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)", ANS X9.62-2005, November 2005.

9.2. Informative References

[BonehShoup] Boneh, D. and V. Shoup, "A Graduate Course in Applied Cryptography", January 2020, <http://toc.cryptobook.us/book.pdf>.

[FROST20] Komlo, C. and I. Goldberg, "Two-Round Threshold Signatures with FROST", 22 December 2020, <https://eprint.iacr.org/2020/852.pdf>.

[Pornin22] Pornin, T., "Point-Halving and Subgroup Membership in Twisted Edwards Curves", 6 September 2022, <https://eprint.iacr.org/2022/1164.pdf>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC

4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/rfc/rfc4086>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/rfc/rfc7748>.

[Schnorr21] Crites, E., Komlo, C., and M. Maller, "How to Prove Schnorr Assuming Schnorr", 11 October 2021, <https://eprint.iacr.org/2021/1375>.

[StrongerSec22] Bellare, M., Tessaro, S., and C. Zhu, "Stronger Security for Non-Interactive Threshold Signatures: BLS and FROST", 1 June 2022, <https://eprint.iacr.org/2022/833>.

[TLS] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

Appendix A. Acknowledgments

This document was improved based on input and contributions by the Zcash Foundation engineering team. In addition, the authors of this document would like to thank Isis Lovecruft, Alden Torres, T. Wilson-Brown, and Conrado Gouvea for their inputs and contributions.

Appendix B. Schnorr Signature Generation and Verification for Prime-Order Groups

This section contains descriptions of functions for generating and verifying Schnorr signatures. It is included to complement the routines present in [RFC8032] for prime-order groups, including ristretto255, P-256, and secp256k1. The functions for generating and verifying signatures are prime_order_sign and prime_order_verify, respectively.

The function prime_order_sign produces a Schnorr signature over a message given a full secret signing key as input (as opposed to a key share.)

```
  prime_order_sign(msg, sk):
```

```
  Inputs:
  - msg, message to sign, a byte string
  - sk, secret key, a Scalar

  Outputs: (R, z), a Schnorr signature consisting of an Element R and Sc

  def prime_order_sign(msg, sk):
    r = G.RandomScalar()
    R = G.ScalarBaseMult(r)
    PK = G.ScalarBaseMult(sk)
    comm_enc = G.SerializeElement(R)
    pk_enc = G.SerializeElement(PK)
    challenge_input = comm_enc || pk_enc || msg
    c = H2(challenge_input)
    z = r + (c * sk) // Scalar addition and multiplication
    return (R, z)
```

The function prime_order_verify verifies Schnorr signatures with validated inputs. Specifically, it assumes that signature R component and public key belong to the prime-order group.

```
  prime_order_verify(msg, sig, PK):

  Inputs:
  - msg, signed message, a byte string
  - sig, a tuple (R, z) output from signature generation
  - PK, public key, an Element

  Outputs: 1 if signature is valid, and 0 otherwise

  def prime_order_verify(msg, sig = (R, z), PK):
    comm_enc = G.SerializeElement(R)
    pk_enc = G.SerializeElement(PK)
    challenge_input = comm_enc || pk_enc || msg
    c = H2(challenge_input)

    l = G.ScalarBaseMult(z)
    r = R + G.ScalarMult(PK, c)
    return l == r
```

Appendix C. Trusted Dealer Key Generation

One possible key generation mechanism is to depend on a trusted dealer, wherein the dealer generates a group secret s uniformly at random and uses Shamir and Verifiable Secret Sharing as described in Appendix C.1 and Appendix C.2 to create secret shares of s, denoted s_i for i = 0, ..., MAX_PARTICIPANTS, to be sent to all

MAX_PARTICIPANTS participants. This operation is specified in the trusted_dealer_keygen algorithm. The mathematical relation between the secret key s and the MAX_SIGNER secret shares is formalized in the secret_share_combine(shares) algorithm, defined in Appendix C.1.

The dealer that performs trusted_dealer_keygen is trusted to 1) generate good randomness, and 2) delete secret values after distributing shares to each participant, and 3) keep secret values confidential.

Inputs:
- secret_key, a group secret, a Scalar, that MUST be derived from at l
- MAX_PARTICIPANTS, the number of shares to generate, an integer
- MIN_PARTICIPANTS, the threshold of the secret sharing scheme, an int

Outputs:
- participant_private_keys, MAX_PARTICIPANTS shares of the secret key consisting of the participant identifier and the key share (a Scalar
- group_public_key, public key corresponding to the group signing key, Element in G.
- vss_commitment, a vector commitment of Elements in G, to each of the in the polynomial defined by secret_key_shares and whose first eleme G.ScalarBaseMult(s).

```
def trusted_dealer_keygen(secret_key, MAX_PARTICIPANTS, MIN_PARTICIPAN
  # Generate random coefficients for the polynomial
  coefficients = []
  for i in range(0, MIN_PARTICIPANTS - 1):
    coefficients.append(G.RandomScalar())
  participant_private_keys, coefficients = secret_share_shard(secret_k
  vss_commitment = vss_commit(coefficients):
  return participant_private_keys, vss_commitment[0], vss_commitment
```

It is assumed the dealer then sends one secret key share to each of the NUM_PARTICIPANTS participants, along with vss_commitment. After receiving their secret key share and vss_commitment, participants MUST abort if they do not have the same view of vss_commitment. Otherwise, each participant MUST perform vss_verify(secret_key_share_i, vss_commitment), and abort if the check fails. The trusted dealer MUST delete the secret_key and secret_key_shares upon completion.

Use of this method for key generation requires a mutually authenticated secure channel between the dealer and participants to send secret key shares, wherein the channel provides confidentiality and integrity. Mutually authenticated TLS is one possible deployment option.

C.1. Shamir Secret Sharing

In Shamir secret sharing, a dealer distributes a secret Scalar s to n participants in such a way that any cooperating subset of MIN_PARTICIPANTS participants can recover the secret. There are two basic steps in this scheme: (1) splitting a secret into multiple shares, and (2) combining shares to reveal the resulting secret.

This secret sharing scheme works over any field F. In this specification, F is the scalar field of the prime-order group G.

The procedure for splitting a secret into shares is as follows.

```
secret_share_shard(s, MAX_PARTICIPANTS, MIN_PARTICIPANTS):

Inputs:
- s, secret value to be shared, a Scalar
- coefficients, an array of size MIN_PARTICIPANTS - 1 with randomly ge
  Scalars, not including the 0th coefficient of the polynomial
- MAX_PARTICIPANTS, the number of shares to generate, an integer less
- MIN_PARTICIPANTS, the threshold of the secret sharing scheme, an int

Outputs:
- secret_key_shares, A list of MAX_PARTICIPANTS number of secret share
  consisting of the participant identifier and the key share (a Scalar
- coefficients, a vector of MIN_PARTICIPANTS coefficients which unique

Errors:
- "invalid parameters", if MIN_PARTICIPANTS > MAX_PARTICIPANTS or if M

def secret_share_shard(s, coefficients, MAX_PARTICIPANTS, MIN_PARTICIP
  if MIN_PARTICIPANTS > MAX_PARTICIPANTS:
    raise "invalid parameters"
  if MIN_PARTICIPANTS < 2:
    raise "invalid parameters"

  # Prepend the secret to the coefficients
  coefficients = [s] + coefficients

  # Evaluate the polynomial for each point x=1,...,n
  secret_key_shares = []
  for x_i in range(1, MAX_PARTICIPANTS + 1):
    y_i = polynomial_evaluate(Scalar(x_i), coefficients)
    secret_key_share_i = (x_i, y_i)
    secret_key_share.append(secret_key_share_i)
  return secret_key_shares, coefficients
```

Let points be the output of this function. The i-th element in points is the share for the i-th participant, which is the randomly generated polynomial evaluated at coordinate i. We denote a secret

share as the tuple (i, points[i]), and the list of these shares as shares. i MUST never equal 0; recall that f(0) = s, where f is the polynomial defined in a Shamir secret sharing operation.

The procedure for combining a shares list of length MIN_PARTICIPANTS to recover the secret s is as follows; the algorithm polynomial_interpolation is defined in {{dep-polynomial-interpolate}}.

```
secret_share_combine(shares):

Inputs:
- shares, a list of at minimum MIN_PARTICIPANTS secret shares, each a
  where i and f(i) are Scalars

Outputs: The resulting secret s, a Scalar, that was previously split i

Errors:
- "invalid parameters", if fewer than MIN_PARTICIPANTS input shares ar

def secret_share_combine(shares):
  if len(shares) < MIN_PARTICIPANTS:
    raise "invalid parameters"
  s = polynomial_interpolation(shares)
  return s
```

C.1.1. Deriving the constant term of a polynomial

Secret sharing requires "splitting" a secret, which is represented as a constant term of some polynomial f of degree t-1. Recovering the constant term occurs with a set of t points using polynomial interpolation, defined as follows.

```
Inputs:
- points, a set of t distinct points on a polynomial f, each a tuple o
  Scalar values representing the x and y coordinates

Outputs: The constant term of f, i.e., f(0)

def polynomial_interpolation(points):
  x_coords = []
  for point in points:
    x_coords.append(point.x)

  f_zero = Scalar(0)
  for point in points:
    delta = point.y * derive_lagrange_coefficient(point.x, x_coords)
    f_zero = f_zero + delta

  return f_zero
```

C.2. Verifiable Secret Sharing

Feldman's Verifiable Secret Sharing (VSS) builds upon Shamir secret sharing, adding a verification step to demonstrate the consistency of a participant's share with a public commitment to the polynomial f for which the secret s is the constant term. This check ensures that all participants have a point (their share) on the same polynomial, ensuring that they can later reconstruct the correct secret.

The procedure for committing to a polynomial f of degree at most MIN_PARTICIPANTS-1 is as follows.

vss_commit(coeffs):

Inputs:
- coeffs, a vector of the MIN_PARTICIPANTS coefficients which uniquely a polynomial f.

Outputs: a commitment vss_commitment, which is a vector commitment to coefficients in coeffs, where each element of the vector commitment is

```
def vss_commit(coeffs):
  vss_commitment = []
  for coeff in coeffs:
    A_i = G.ScalarBaseMult(coeff)
    vss_commitment.append(A_i)
  return vss_commitment
```

The procedure for verification of a participant's share is as follows. If vss_verify fails, the participant MUST abort the protocol, and failure should be investigated out of band.

```
vss_verify(share_i, vss_commitment):

Inputs:
- share_i: A tuple of the form (i, sk_i), where i indicates the partic
  identifier, and sk_i the participant's secret key, a secret share of
  constant term of f, where sk_i is a Scalar.
- vss_commitment: A VSS commitment to a secret polynomial f, a vector
  to each of the coefficients in coeffs, where each element of the vec
  is an Element

Outputs: 1 if sk_i is valid, and 0 otherwise

vss_verify(share_i, vss_commitment)
  (i, sk_i) = share_i
  S_i = ScalarBaseMult(sk_i)
  S_i' = G.Identity()
  for j in range(0, MIN_PARTICIPANTS):
    S_i' += G.ScalarMult(vss_commitment[j], pow(i, j))
  if S_i == S_i':
    return 1
  return 0

 We now define how the Coordinator and participants can derive group
 info, which is an input into the FROST signing protocol.

  derive_group_info(MAX_PARTICIPANTS, MIN_PARTICIPANTS, vss_commitment

  Inputs:
  - MAX_PARTICIPANTS, the number of shares to generate, an integer
  - MIN_PARTICIPANTS, the threshold of the secret sharing scheme, an i
  - vss_commitment: A VSS commitment to a secret polynomial f, a vecto
  coefficients in coeffs, where each element of the vector commitment

  Outputs:
  - PK, the public key representing the group, an Element.
  - participant_public_keys, a list of MAX_PARTICIPANTS public keys PK
    where each PK_i is the public key, an Element, for participant i.

  derive_group_info(MAX_PARTICIPANTS, MIN_PARTICIPANTS, vss_commitment
    PK = vss_commitment[0]
    participant_public_keys = []
    for i in range(1, MAX_PARTICIPANTS+1):
      PK_i = G.Identity()
      for j in range(0, MIN_PARTICIPANTS):
        PK_i += G.ScalarMult(vss_commitment[j], pow(i, j))
      participant_public_keys.append(PK_i)
    return PK, participant_public_keys
```

Appendix D. Random Scalar Generation

Two popular algorithms for generating a random integer uniformly distributed in the range [0, G.Order() -1] are as follows:

D.1. Rejection Sampling

Generate a random byte array with Ns bytes, and attempt to map to a Scalar by calling DeserializeScalar in constant time. If it succeeds, return the result. If it fails, try again with another random byte array, until the procedure succeeds. Failure to implement this in constant time can leak information about the underlying corresponding Scalar.

Note the that the Scalar size might be some bits smaller than the array size, which can result in the loop iterating more times than required. In that case it's acceptable to set the high-order bits to 0 before calling DeserializeScalar, but care must be taken to not set to zero more bits than required. For example, in the FROST(Ed25519, SHA-512) ciphersuite, the order has 253 bits while the array has 256; thus the top 3 bits of the last byte can be set to zero.

D.2. Wide Reduction

Generate a random byte array with l = ceil(((3 * ceil(log2(G.Order()))) / 2) / 8) bytes, and interpret it as an integer; reduce the integer modulo G.Order() and return the result. See Section 5 of [HASH-TO-CURVE] for the underlying derivation of l.

Appendix E. Test Vectors

This section contains test vectors for all ciphersuites listed in Section 6. All Element and Scalar values are represented in serialized form and encoded in hexadecimal strings. Signatures are represented as the concatenation of their constituent parts. The input message to be signed is also encoded as a hexadecimal string.

Each test vector consists of the following information.

- Configuration. This lists the fixed parameters for the particular instantiation of FROST, including MAX_PARTICIPANTS, MIN_PARTICIPANTS, and NUM_PARTICIPANTS.

- Group input parameters. This lists the group secret key and shared public key, generated by a trusted dealer as described in Appendix C, as well as the input message to be signed. The randomly generated coefficients produced by the trusted dealer to share the group signing secret are also listed. Each coefficient is identified by its index, e.g.,

share_polynomial_coefficients[1] is the coefficient of the first term in the polynomial. Note that the 0-th coefficient is omitted as this is equal to the group secret key. All values are encoded as hexadecimal strings.

* Signer input parameters. This lists the signing key share for each of the NUM_PARTICIPANTS participants.

* Round one parameters and outputs. This lists the NUM_PARTICIPANTS participants engaged in the protocol, identified by their integer identifier, and for each participant: the hiding and binding commitment values produced in Section 5.1; the randomness values used to derive the commitment nonces in nonce_generate; the resulting group binding factor input computed in part from the group commitment list encoded as described in Section 4.3; and group binding factor as computed in Section 5.2).

* Round two parameters and outputs. This lists the NUM_PARTICIPANTS participants engaged in the protocol, identified by their integer identifier, along with their corresponding output signature share as produced in Section 5.2.

* Final output. This lists the aggregate signature as produced in Section 5.3.

E.1. FROST(Ed25519, SHA-512)

```
// Configuration information
MAX_PARTICIPANTS: 3
MIN_PARTICIPANTS: 2
NUM_PARTICIPANTS: 2

// Group input parameters
group_secret_key: 7b1c33d3f5291d85de664833beb1ad469f7fb6025a0ec78b3a7
90c6e13a98304
group_public_key: 15d21ccd7ee42959562fc8aa63224c8851fb3ec85a3faf66040
d380fb9738673
message: 74657374
share_polynomial_coefficients[1]: 178199860edd8c62f5212ee91eff1295d0d
670ab4ed4506866bae57e7030b204

// Signer input parameters
P1 participant_share: 929dcc590407aae7d388761cddb0c0db6f5627aea8e217f
4a033f2ec83d93509
P2 participant_share: a91e66e012e4364ac9aaa405fcafd370402d9859f7b6685
c07eed76bf409e80d
P3 participant_share: d3cb090a075eb154e82fdb4b3cb507f110040905468bb9c
46da8bdea643a9a02

// Round one parameters
participant_list: 1,3

// Signer round one outputs
P1 hiding_nonce_randomness: 5c8f4d74076647069566b460154525c0d80bf278a
49ef0da7e5b56ab1cceac08
P1 binding_nonce_randomness: 90cda7316703b5f5e0c890ac456a1f64c0db1e9b
025493b50f795a38ec373b69
P1 hiding_nonce: 8fa57cf7b4f6476d1f0ee1ee972ea9acc79633fc6a46a1ab7223
4e8f4cc65b00
P1 binding_nonce: 305458d8109e95c71d5e61fb985a8e00126276dcd02ba9814fa
2a5674f36f70d
P1 hiding_nonce_commitment: 9edff055e77f1b7addf4bf48a58beea392a0032ac
f2744196c61f3b3c3b1cb47
P1 binding_nonce_commitment: 4a03fb80acea2712f2ff1548e990457cdd58fb17
1a69f70fcdf53d1b0ea83008
P1 binding_factor_input: 25120720c3a416e292edfb0510780bc84eb734347a5f
d84dd46d0dcbf3a21d21a23a776628859678a968acc8c8564c4641a1fd4b29a12d536
7ca12ab10b6b497c2385b35930a34d98238021b25ab0641e39f8cf4277f90e3c03161
b1755e7b08b7c5e5334a57e89c5e841186e9cca95214851ed99dcbd07f104fc399194
fd8dc0100000000000000000000000000000000000000000000000000000000000000
P1 binding_factor: febb7ae5b20ffe3c1d2bcf08f7e6104a78b3136ad3b7df35d5
f84878ce7bb50d
P3 hiding_nonce_randomness: a85d5ce57b709e59b07b1fefc80a50a928b409881
7e482fa38f831ab453da14d
P3 binding_nonce_randomness: b74a04b3a7e7311c974204c1e142952d1e93cb1e
e0791c90126d2c4eb375e0e1
```

P3 hiding_nonce: 59b0da2f402a69956b4c904737afefe29ef0aaa4401330f56dec
ebe450204a08
P3 binding_nonce: 1a54642c6b975f1b39a1b391bebc4f72b6dc6ca551cbebedee6
ce5a60069fd0d
P3 hiding_nonce_commitment: 5e11b65802153b03332533a3018d22fe760098508
a250832a457e6004848411b
P3 binding_nonce_commitment: 9ef535b69175da3833d330c1a43823b4b08dc1fb
38fdc3af99f6804d8b250a09
P3 binding_factor_input: 25120720c3a416e292edfb0510780bc84eb734347a5f
d84dd46d0dcbf3a21d21a23a776628859678a968acc8c8564c4641a1fd4b29a12d536
7ca12ab10b6b497c2385b35930a34d98238021b25ab0641e39f8cf4277f90e3c03161
b1755e7b08b7c5e5334a57e89c5e841186e9cca95214851ed99dcbd07f104fc399194
fd8dc03000000000000000000000000000000000000000000000000000000000000
P3 binding_factor: 6662bfe16e045aba72954ef54b6dabfc61db322698b57d5a40
94b39d4ce01f05

// Round two parameters
participant_list: 1,3

// Signer round two outputs
P1 sig_share: d21fd9d6f793c8efef622ae139ec94071dc1b1f47624f638f4f8615
e1934bc09
P3 sig_share: 7b599a881fc39829b2f69b194fd64bd6ee0df8b34e3d067be5612fa
c4da8ac0b

sig: 7c64be7528effeab65503e813ee28f94c53a4d8e45a0a71e936c840c78e42233
60a57d02fdf34ec1cbbcce57aac801c90bcfa9a8c561fcb3d95a910a67dc6805

E.2. FROST(Ed448, SHAKE256)

```
// Configuration information
MAX_PARTICIPANTS: 3
MIN_PARTICIPANTS: 2
NUM_PARTICIPANTS: 2

// Group input parameters
group_secret_key: 6298e1eef3c379392caaed061ed8a31033c9e9e3420726f23b4
04158a401cd9df24632adfe6b418dc942d8a091817dd8bd70e1c72ba52f3c00
group_public_key: 3832f82fda00ff5365b0376df705675b63d2a93c24c6e81d408
01ba265632be10f443f95968fadb70d10786827f30dc001c8d0f9b7c1d1b000
message: 74657374
share_polynomial_coefficients[1]: dbd7a514f7a731976620f0436bd135fe8dd
dc3fadd6e0d13dbd58a1981e587d377d48e0b7ce4e0092967c5e85884d0275a7a740b
6abdcd0500

// Signer input parameters
P1 participant_share: 4a2b2f5858a932ad3d3b18bd16e76ced3070d72fd79ae44
02df201f525e754716a1bc1b87a502297f2a99d89ea054e0018eb55d39562fd0100
P2 participant_share: 2503d56c4f516444a45b080182b8a2ebbe4d9b2ab509f25
308c88c0ea7ccdc44e2ef4fc4f63403a11b116372438a1e287265cadeff1fcb0700
P3 participant_share: 00db7a8146f995db0a7cf844ed89d8e94c2b5f259378ff6
6e39d172828b264185ac4decf7219e4aa4478285b9c0eef4fccdf3eea69dd980d00

// Round one parameters
participant_list: 1,3

// Signer round one outputs
P1 hiding_nonce_randomness: b5eb1b4e369491e7d09427b01285803215bbc4c44
0edebf8a337bfa3bc75a931
P1 binding_nonce_randomness: c73b102813418576607ed61e726233f5df303f96
38af479b05fbc8dda6739eeb
P1 hiding_nonce: aa4606f9b2e5cf331e473833087769579685c4582a80ae45b01e
3ffe83c0455b6f0eb05b320db0de0223509e8f8d30f356173b1ff2383a0100
P1 binding_nonce: 90c118161d17804126d474af8cc1e2358319b704b8adb4088a8
a36a3f48e8ec1c6b5106d394a36bd9e62d7dd7ee2a9978c866931d8da792300
P1 hiding_nonce_commitment: 55d2fe1b9b91d8bd8511417ff9b315c752944acdf
a483f134d2004c441bf6da055219dd0632ccaf8f35970fac9bea116f7269855833453
9800
P1 binding_nonce_commitment: 8ac0e4cf4d9fb584d5372d963341d6144d5d0310
b6029c34c7c7a2bba4b259a3bb803da16e024764de4612aafc8d63d48f4e2a97df3f4
0e800
P1 binding_factor_input: 766a004ac6e87a2fa70f2095b19596ac33b94e2f6803
e1a5b8fa8ea5adaf3e7989b2c167a38a42a1693ad69cfd674e089a498672753563d53
54654ba106d5fdffb134a8917fae91d412164436f734b95572af6208605744400c6ff
9a60fa2ce8fb7f3213414c32e347ee2e29e3d17654ef0277d99dbfcd70bf9162c46ca
7f45754d1eabb8ab239f4fb88cc8af91e5f1af5d81f1f2f8dfb564c6ddb0a75bdf681
b2f403d7d903000c647602978b2e714360a3847ca469b932d4dcdff6ea421c27a382c
c230625e7c9b45e662056ef73687412f705f77fc4bc20e3d902888c02c2fd2ff07801
0000000000000000000000000000000000000000000000000000000000000000000
```

```
0000000000000000000000000000000000000000000000000
P1 binding_factor: 4d58a6a888309da8b8d2d2068a44ad3c597591b1b806cc8ac1
2d2c1202c66ae9c4e68b6bfd2ef3b3a5968a288fa5a2aed86161ebeb2aa60e00
P3 hiding_nonce_randomness: 516a0e41f6b2f1f9a6661b5549db7486c6073249b
6dcc5bbdca69e7fd6bcbd12
P3 binding_nonce_randomness: 28f89d1c037cf61ccd08f7e06fa2289bb5a31a60
5455bad6bec61923d60fe811
P3 hiding_nonce: 4fd3566260ecc3f57f1cfb7c84724d2fc80f90dab5ee3613eb9b
278e71fd9d88c8dae06ec1bbf2fd535ec3fca393dd011d63ee5c3db7383000
P3 binding_nonce: c0a583112b13e238f249bac41a01ccb1ad679c4feaab14fc7ab
0df3511066cd117fadfc1ec3a0fd31820e8857943675262ebc5d16194cd2300
P3 hiding_nonce_commitment: 148325fe0941216938cd6bcbe45a68c3184b1f087
1f2b17f3bbdc4e2aef9c03d27845bd6f45c05a810b0dd2582af3e11518015405a4ffb
d480
P3 binding_nonce_commitment: 2a34a68cfb05f20f0e2437daf4480cbafdab3243
9a6798c1f7b36e0e47714c570242ef8972914ec65b8ed49784dc62fbdc8f7838faa75
63700
P3 binding_factor_input: 766a004ac6e87a2fa70f2095b19596ac33b94e2f6803
e1a5b8fa8ea5adaf3e7989b2c167a38a42a1693ad69cfd674e089a498672753563d53
54654ba106d5fdffb134a8917fae91d412164436f734b95572af6208605744400c6ff
9a60fa2ce8fb7f3213414c32e347ee2e29e3d17654ef0277d99dbfcd70bf9162c46ca
7f45754d1eabb8ab239f4fb88cc8af91e5f1af5d81f1f2f8dfb564c6ddb0a75bdf681
b2f403d7d903000c647602978b2e714360a3847ca469b932d4dcdff6ea421c27a382c
c230625e7c9b45e662056ef73687412f705f77fc4bc20e3d902888c02c2fd2ff07803
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000
P3 binding_factor: 7bfb1317cd530ba478544ee911687d0d0fa5e4c49e40a79347
2601f9c587440478eebb90bbec84c98055783a32e442a3b40503b2a9389d3d00

// Round two parameters
participant_list: 1,3

// Signer round two outputs
P1 sig_share: d982cdbb5d14fb53c8cb28cfb690ec38fc4f8f44c666de910703ee7
ba2183989286c614c9bb5319a3a70bf22ee68ca54b3a6b90f33a33a1a00
P3 sig_share: adc2d8301b5f5597af72c51cce53ab6d265e8e788ce4738b1e2fbdc
e24eaa283b0bfbd7b587a407844c20e958c21a5ba9895b51fdfbb3e3700

sig: 22943fef7107155064b0baa1d6ad78ee6c5d909ca88ac8bd1002bde99e124a34
514f826d24d83fbdecf7752be1fd166fd31752fd4eb3a3280093004e41e6b0d7c722a
f285e12222b859277470e097003593c0ee1cdc702dc0cd92b1fc8f32f72127f32ceb7
7a8a6f0f4c3c6f2f125f791100
```

E.3. FROST(ristretto255, SHA-512)

```
// Configuration information
MAX_PARTICIPANTS: 3
MIN_PARTICIPANTS: 2
NUM_PARTICIPANTS: 2

// Group input parameters
group_secret_key: 1b25a55e463cfd15cf14a5d3acc3d15053f08da49c8afcf3ab2
65f2ebc4f970b
group_public_key: e2a62f39eede11269e3bd5a7d97554f5ca384f9f6d3dd9c3c0d
05083c7254f57
message: 74657374
share_polynomial_coefficients[1]: 410f8b744b19325891d73736923525a4f59
6c805d060dfb9c98009d34e3fec02

// Signer input parameters
P1 participant_share: 5c3430d391552f6e60ecdc093ff9f6f4488756aa6cebdba
d75a768010b8f830e
P2 participant_share: b06fc5eac20b4f6e1b271d9df2343d843e1e1fb03c4cbb6
73f2872d459ce6f01
P3 participant_share: f17e505f0e2581c6acfe54d3846a622834b5e7b50cad9a2
109a97ba7a80d5c04

// Round one parameters
participant_list: 1,3

// Signer round one outputs
P1 hiding_nonce_randomness: b8f4ed5c201cb19151d091945eb0011db6bf4b816
debfc21fc6833a99dca87b8
P1 binding_nonce_randomness: 2588621bdf623b8a687f4def4f4f720b8da24f6b
15c64a2a11dc74ffa04c9c02
P1 hiding_nonce: 05a9f136274e02107502f5073e91b72bd42165f277d524b2eb3f
9c6fa7a38a08
P1 binding_nonce: 8fe91bee76721535b6ac5b84ea5474892cb476404217a9b6129
d1974cd211703
P1 hiding_nonce_commitment: 4ce0c9d3de33b7876122a3490cef6a5b8cc22acc2
9e96195629b5b8cfb1b784f
P1 binding_nonce_commitment: 26d80e82877fd0fd78a803996bf4847ee9d1ffd8
0072c9f16d34c5dde4138115
P1 binding_factor_input: fe9082dcc1ae1ae11380ac4cf0b6e2770af565ff5af9
016254dc7c9d4869cbae0f6e4d94b23e5781b91bc74a25e0c773446b2640290d07c83
f0b067ff870a801f846633bc5505172ee8dc7bb109916b61ea54ddc715684bb50f612
8cec164ab16da6bcebddb83b78493ef85fca52ca460199609523f3eb877d46c399d28
a009001000000000000000000000000000000000000000000000000000000000000
P1 binding_factor: 38915e5ab3b90122785515aa89d20c8e592de9f04cb0f42c6a
e7cf99d3990d0b
P3 hiding_nonce_randomness: 7f417eadf8651b6043eece2965411bfb237b0a468
9c88a8469a2a66b79a6d7f9
P3 binding_nonce_randomness: 53dd06d89cdeb574fc32e83fa138f73f668a99aa
ccfa03f82e68ff6b845b7a8a
```

P3 hiding_nonce: 2efc0a895c5788cb3b65f1032f8d64cf59ed71637353a7103fce
ea669a169f04
P3 binding_nonce: 6e60f3eac7359a4c19a22b618aa4531bd49a1ec83d2c3f503d7
5474ddb15b406
P3 hiding_nonce_commitment: 064f733cb26f728ef185d40e75c50b717cd543abe
8057e38a96831a7de47b17f
P3 binding_nonce_commitment: 6a47acc247a4cbf5b753a1d1f2dcb43241f2df71
3468bf9a7a00e6257442bc46
P3 binding_factor_input: fe9082dcc1ae1ae11380ac4cf0b6e2770af565ff5af9
016254dc7c9d4869cbae0f6e4d94b23e5781b91bc74a25e0c773446b2640290d07c83
f0b067ff870a801f846633bc5505172ee8dc7bb109916b61ea54ddc715684bb50f612
8cec164ab16da6bcebddb83b78493ef85fca52ca460199609523f3eb877d46c399d28
a009003000000000000000000000000000000000000000000000000000000000000000
P3 binding_factor: 3f83b7811980bb032a3f85d857af755a00f4b16a10cdf88f32
d13ea908d3b107

// Round two parameters
participant_list: 1,3

// Signer round two outputs
P1 sig_share: 86879530bc8d700ec20e6211d0c8d27d1b518f5aed00b9eef1b04ed
625c1ab05
P3 sig_share: 66f01933240c000ab4abb203f49bf8bb7138ffe17d5cd6c943ea30c
f9f32410c

sig: 8adca3c885d6ffa07429de8df9ba9bc72a3b0dfa0f6139035deadc72e13c645e
ffa3b906c6365ec09f1d1d72e56aec248d898e3c6b5d8fb8359b7fa5c5f3ec01

E.4. FROST(P-256, SHA-256)

```
// Configuration information
MAX_PARTICIPANTS: 3
MIN_PARTICIPANTS: 2
NUM_PARTICIPANTS: 2

// Group input parameters
group_secret_key: 8ba9bba2e0fd8c4767154d35a0b7562244a4aaf6f36c8fb8735
fa48b301bd8de
group_public_key: 023a309ad94e9fe8a7ba45dfc58f38bf091959d3c99cfbd02b4
dc00585ec45ab70
message: 74657374
share_polynomial_coefficients[1]: 80f25e6c0709353e46bfbe882a11bdbb1f8
097e46340eb8673b7e14556e6c3a4

// Signer input parameters
P1 participant_share: 0c9c1a0fe806c184add50bbdcac913dda73e482daf95dcb
9f35dbb0d8a9f7731
P2 participant_share: 8d8e787bef0ff6c2f494ca45f4dad198c6bee01212d6c84
067159c52e1863ad5
P3 participant_share: 0e80d6e8f6192c003b5488ce1eec8f5429587d48cf00154
1e713b2d53c09d928

// Round one parameters
participant_list: 1,3

// Signer round one outputs
P1 hiding_nonce_randomness: 3c3cce19a46e2539a2a3f0fc1302c04d643c80b61
beff7545f09bf3379ce763f
P1 binding_nonce_randomness: 7550f8224653dec3179586846a156e08911cad54
e785e7a8db13539ad1fd7ac6
P1 hiding_nonce: f07d7e697f7db6b0eaa29e8303e718440197ff47e1918cd4e630
c202ab323223
P1 binding_nonce: 248eb6983093337c1dae193643e8e4618cdafd189fb8289239b
1b54a1778e0cf
P1 hiding_nonce_commitment: 02cc0ef1b65d7284c0be9c555e9877fac2dad008d
f7df6133928f900ca34d82bbe
P1 binding_nonce_commitment: 0281413660fa8a78ae1ca4a65fdad3259246e19e
74f81687b9812246c8b174b1f2
P1 binding_factor_input: 3617acb73b44df565fbcbbbd1824142c473ad1d6c800
7c4b72a298d1eaae57664a317d123a18b12922413ebd80944a0816821b2694b4aea22
16ebbe38eb2daf300000000000000000000000000000000000000000000000000000
0000000001
P1 binding_factor: 0f2c1ea6da0fceb9cbb63bfe81d80bd22c434a5d9810b1cc97
7f7eab7f0e9da4
P3 hiding_nonce_randomness: 89c096201f0aeb36b0b8a33763ac92df1169ead49
fcf2dec8209f858852a345f
P3 binding_nonce_randomness: 8785c0b2a5817cf40c978e28f7804eff570c3c1c
f9df11b37496c318fa38d107
P3 hiding_nonce: dc34074d1f8f427ee8eabc546cb9931cc44371be94ad66f7a27f
```

dc4a665db373
P3 binding_nonce: 0a3e836f36b4011be4f6ef111596838fdfccf89c431d66da2ca
75ea55dcf3b44
P3 hiding_nonce_commitment: 02261bf8649d9943d5f664694219742e9abc501c6
7be6b112e833b144462a662e3
P3 binding_nonce_commitment: 03603f38cefae47af6be8dbeeef53e7b27b5af8d
4159b968d489a1f7295581b0e5
P3 binding_factor_input: 3617acb73b44df565fbcbbbd1824142c473ad1d6c800
7c4b72a298d1eaae57664a317d123a18b12922413ebd80944a0816821b2694b4aea22
16ebbe38eb2daf300000000000000000000000000000000000000000000000000000
0000000003
P3 binding_factor: 7598b94e95da5f81f9b3ad0c40499526429ec80de9cc0a8240
0bbbd2725ea947

// Round two parameters
participant_list: 1,3

// Signer round two outputs
P1 sig_share: 35e4cfe05008a064ad66f2ab62005632b5c44d4daea5cf1ffd193c5
56ec4ddad
P3 sig_share: 79aa1cc540bacfc5a96d4eb1940aa85ec56ec880885748c3de11ff3
55d4e77cf

sig: 03fe31f09eb37b31956749b77eb3d5cf88ac6c2192481e2eaa6e729d4ed8ba0f
28af8eeca590c3702a56d4415cf60afe917b3315ce36fd17e3db2b3b8acc13557c

E.5. FROST(secp256k1, SHA-256)

```
// Configuration information
MAX_PARTICIPANTS: 3
MIN_PARTICIPANTS: 2
NUM_PARTICIPANTS: 2

// Group input parameters
group_secret_key: 0d004150d27c3bf2a42f312683d35fac7394b1e9e318249c1bf
e7f0795a83114
group_public_key: 02f37c34b66ced1fb51c34a90bdae006901f10625cc06c4f646
63b0eae87d87b4f
message: 74657374
share_polynomial_coefficients[1]: fbf85eadae3058ea14f19148bb72b45e439
9c0b16028acaf0395c9b03c823579

// Signer input parameters
P1 participant_share: 08f89ffe80ac94dcb920c26f3f46140bfc7f95b493f8310
f5fc1ea2b01f4254c
P2 participant_share: 04f0feac2edcedc6ce1253b7fab8c86b856a797f44d83d8
2a385554e6e401984
P3 participant_share: 00e95d59dd0d46b0e303e500b62b7ccb0e555d49f5b849f
5e748c071da8c0dbc

// Round one parameters
participant_list: 1,3

// Signer round one outputs
P1 hiding_nonce_randomness: cfeb82944dbbddde7863519b623acdaf0962217ae
fa5a4ad08b07f176600c070
P1 binding_nonce_randomness: 11b45cbcc8eec8877741fc2622ff0c7fbf68a6bd
9b3f8ee313150d6e0bcb5d83
P1 hiding_nonce: b97f0b3fa40a05cbbda2d0acce6170f046520bbf884748e1949b
007c6f11b410
P1 binding_nonce: 6f0c660fb917a39a8b61893ebf8d5e6000da5e766efe717bf40
20c444327a4a0
P1 hiding_nonce_commitment: 03e030abe3c24a6258fe5b15abcdfbb539f146998
d86396e2d3f3cf6d5198f60de
P1 binding_nonce_commitment: 0264617bf0828fa5eadc831b2fa4b4038d7e39c7
01708282d707d5f58e6e41d28e
P1 binding_factor_input: d759fa818c284537bbb2efa2d7247eac9232b7b992cd
49237106acab251dd954b5398f26122974fbb2a00ebfa06caafe44bc289dc964b83dd
4f46411ebd2bce900000000000000000000000000000000000000000000000000000
0000000001
P1 binding_factor: d80f1df14cfd6803b8501866ca1667297f7867c21886bc04a4
0b56964efa6af6
P3 hiding_nonce_randomness: c8e243917b9d06df6a92c6beca63d6be08325e7b9
29114ec4670f54c07529b47
P3 binding_nonce_randomness: 920d7850ac0d04049eae8e107ee0d86d416d9887
23c505fde8e68ef3e6c36153
P3 hiding_nonce: 9e73856e36f0ac77165af15ced03c71c4b0b3df9e63e1fc550d1
```

```
38effb4cf827
P3 binding_nonce: d1620add724f71f28248822ce5fad517428c6b7247dee8d6a46
56b795d058173
P3 hiding_nonce_commitment: 02ba01e4f0a2ead5da318536893cd20f658a51e78
0dfc96b7604fe8ab73b34a00f
P3 binding_nonce_commitment: 0317ec2be255f4647a98b3573582c48ac5c5f087
fdaa057a92ed363f1a1a642056
P3 binding_factor_input: d759fa818c284537bbb2efa2d7247eac9232b7b992cd
49237106acab251dd954b5398f26122974fbb2a00ebfa06caafe44bc289dc964b83dd
4f46411ebd2bce900000000000000000000000000000000000000000000000000000
0000000003
P3 binding_factor: edd9bb6d85b3610a73e9eaeb2941924b25497d97cfdec17e47
9ef5e9b2a7dcd7

// Round two parameters
participant_list: 1,3

// Signer round two outputs
P1 sig_share: 8a89a17b72e7235d6fbbd0c25db85a1ede389f84d290258c1a42397
dff111626
P3 sig_share: 100275a38484954d345e238f92d4161bf8064a2c4767e916f406ada
b0a734d26

sig: 02652105e8b9d3b30e6868c8cb4cb86d858201fc0f7cef211b6c8f0426d8ac63
f69a8c171ef76bb8aaa419f451f08c703ad63ee9b119f80ea30e48e7290984634c
```

Authors' Addresses

Deirdre Connolly
Zcash Foundation

Email: durumcrustulum@gmail.com

Chelsea Komlo
University of Waterloo, Zcash Foundation

Email: ckomlo@uwaterloo.ca

Ian Goldberg
University of Waterloo

Email: iang@uwaterloo.ca

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net