Workgroup: CFRG
Internet-Draft: draft-irtf-cfrg-frost-12
Published: 24 January 2023
Intended Status: Informational
Expires: 28 July 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

This document specifies the Flexible Round-Optimized Schnorr Threshold (FROST) signing protocol. FROST signatures can be issued after a threshold number of entities cooperate to compute a signature, allowing for improved distribution of trust and redundancy with respect to a secret key. FROST depends only on a prime-order group and cryptographic hash function. This document specifies a number of ciphersuites to instantiate FROST using different prime-order groups and hash functions. One such ciphersuite can be used to produce signatures that can be verified with an Edwards-Curve Digital Signature Algorithm (EdDSA, as defined in RFC8032) compliant verifier. However, unlike EdDSA, the signatures produced by FROST are not deterministic. This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

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

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents 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 28 July 2023.

Copyright Notice

Table of Contents

1. Introduction

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 specifies the Flexible Round-Optimized Schnorr Threshold (FROST) signing protocol based on the original work 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. FROST requires two rounds to compute a

signature. Single-round signing variants based on [FROST20] are out of scope.

FROST depends only on a prime-order group and cryptographic hash function. This document specifies a number of ciphersuites to instantiate FROST using different prime-order groups and hash functions. Two ciphersuites can be used to produce signatures that are compatible with Edwards-Curve Digital Signature Algorithm (EdDSA) variants Ed25519 and Ed448 as specified in [RFC8032], i.e., the signatures can be verified with an [RFC8032] compliant verifier. However, unlike EdDSA, the 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.

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.

This document represents the consensus of the Crypto Forum Research Group (CFRG). It is not an IETF product and is not a standard.

1.1. Change Log

draft-12

- Address RGLC feedback (#399, #396, #395, #394, #393, #384, #382, #397, #378, #376, #375, #374, #373, #371, #370, #369, #368, #367, #366, #364, #363, #362, #361, #359, #358, #357, #356, #354, #353, #352, #350, #349, #348, #347, #314)

- Fix bug in signature share serialization (#397)

- Fix various editorial issues (#385)

draft-11

- Update version string constant (#307)

- Make SerializeElement reject the identity element (#306)

- Make ciphersuite requirements explicit (#302)

- Fix various editorial issues (#303, #301, #299, #297)

draft-10

- Update version string constant (#296)

- Fix some editorial issues from Ian Goldberg (#295)

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 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 result, a Scalar, of a to the power of b, e.g., pow(2, 3) = 8 modulo the relevant group order p.

- || 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 the type 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 on 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. It is assumed that group element addition, negation, and equality comparison can be efficiently computed for arbitrary group elements.

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. Moreover, we use the type NonZeroScalar to denote a Scalar value that is not equal to zero, i.e., Scalar(0). We denote equality comparison of these types as == and assignment of values by =. When comparing Scalar values, e.g., for the purposes of sorting lists of Scalar values, the least nonnegative representation mod p is used.

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): Outputs the scalar multiplication between Element A and Scalar k.

- ScalarBaseMult(k): Outputs 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. This function raises an error if A is the identity element of the group.

*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 raises an error if deserialization fails or if 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 raises an error 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 is modeled as a random oracle in security proofs for the protocol (see [FROST20] and [StrongerSec22]). For concrete recommendations on hash functions which SHOULD be used in practice, see Section 6. Using H, we introduce distinct domain-separated hashes, H1, H2, H3, H4, and H5:

*H1, H2, and H3 map arbitrary byte strings to Scalar elements associated with the prime-order group.

*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;

*Polynomials, Section 4.2;

*Encoding operations, Section 4.3;

*Signature binding computation Section 4.4;

*Group commitment computation Section 4.5; and

*Signature challenge computation Section 4.6.

The following sections describe 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 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. Polynomials

This section defines polynomials over Scalars that are used in the main protocol. A polynomial of maximum degree t is represented as a list of t+1 coefficients, where the constant term of the polynomial is in the first position and the highest-degree coefficient is in the last position. For example, the polynomial x^2 + 2x + 3 has degree 2 and is represented as a list of 3 coefficients [3, 2, 1]. A point on the polynomial f is a tuple (x, y), where y = f(x).

The function derive_interpolating_value derives a value used for polynomial interpolation. It is provided a list of x-coordinates as input, each of which cannot equal 0.

```
derive_interpolating_value(x_i, L):

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

Outputs:
- value, a Scalar.

Errors:
- "invalid parameters", if 1) x_i is not in L, or if 2) any
  x-coordinate is represented more than once in L.

def derive_interpolating_value(x_i, L):
  if x_i not in L:
    raise "invalid parameters"
  for x_j in L:
    if count(x_j, 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

  value = numerator / denominator
  return value
```

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 byte string 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 a NonZeroScalar identifier i and two commitment Element va
    (hiding_nonce_commitment_i, binding_nonce_commitment_i). This list M
    in ascending order by identifier.

  Outputs:
  - encoded_group_commitment, the serialized representation of commitmen

  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 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 a NonZeroScalar identifier i and two commitment Element va
    (hiding_nonce_commitment_i, binding_nonce_commitment_i). This list M
    in ascending order by identifier.

  Outputs:
  - identifiers, a list of NonZeroScalar values.

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 a NonZeroScalar identifier i and Scalar binding factor.
  - identifier, participant identifier, a NonZeroScalar.

  Outputs:
  - binding_factor, a Scalar.

  Errors:
  - "invalid participant", when the designated participant is not known.

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 a NonZeroScalar identifier i and two commitment Element va
    (hiding_nonce_commitment_i, binding_nonce_commitment_i). This list M
    in ascending order by identifier.
  - msg, the message to be signed.

  Outputs:
  - binding_factor_list, a list of (NonZeroScalar, Scalar) tuples repres

  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 a NonZeroScalar identifier i and two commitment Element va
  (hiding_nonce_commitment_i, binding_nonce_commitment_i). This list M
  sorted in ascending order by identifier.
- binding_factor_list = [(i, binding_factor), ...],
  a list of (NonZeroScalar, Scalar) tuples representing the binding fa
  for the given identifier.

Outputs:
- group_commitment, an Element.

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_factor_lis
    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, the group commitment, an Element.
- group_public_key, the public key corresponding to the group signing
  Element.
- msg, the message to be signed, a byte string.

Outputs:
- challenge, a Scalar.

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 FROST signing 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 non-zero 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 the Coordinator and the set of signer participants are chosen externally to the protocol. Note that it is possible to deploy the protocol without a distinguished Coordinator; see Section 7.5 for more information.

FROST produces signatures that can be verified as if they were produced from a single signer using a signing key s with corresponding public key PK, where s is a Scalar value and PK = G.ScalarBaseMult(s). As a threshold signing protocol, the group signing key s is Shamir secret-shared amongst each of the MAX_PARTICIPANTS participants and used to produce signatures; see [ShamirSecretSharing] for more information about Shamir secret sharing. In particular, FROST assumes each participant is configured with the following information:

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

- A signing key sk_i, which is a Scalar value representing the i-th Shamir secret share of the group signing key s. In particular, sk_i is the value f(i) on a secret polynomial f of degree (MIN_PARTICIPANTS - 1), where s is f(0). The public key corresponding to this signing key share is PK_i = G.ScalarBaseMult(sk_i).

The Coordinator and each participant are additionally configured with common group information, denoted "group info," which consists of the following:

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

- Public keys PK_i for each participant, 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.

FROST requires two rounds to complete. In the first round, participants generate and publish one-time-use commitments to be used in the second round. In the second round, each participant produces a share of the signature over the Coordinator-chosen message and the other participant commitments. After the second round completes, the Coordinator aggregates the signature shares to produce a final signature. The Coordinator SHOULD abort if the signature is invalid; see Section 5.4 for more information about dealing with invalid signatures and misbehaving participants. This complete interaction, without abort, is shown in Figure 1.

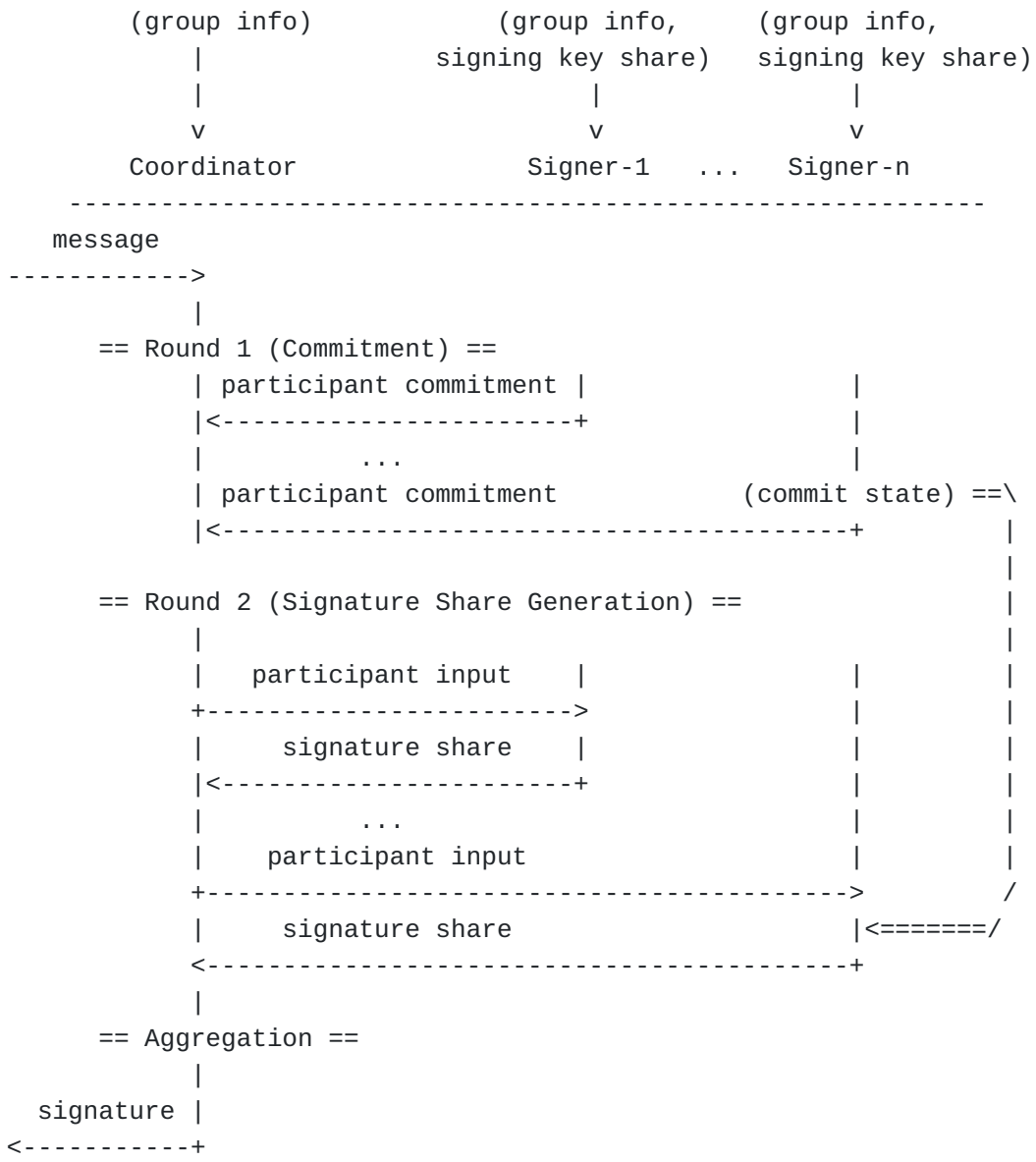

Figure 1: FROST protocol overview

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 the two 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 misbehaving participants, 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 used in more than one invocation of sign, and the nonces 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 its identifier and commitments (from the first round) appear in commitment_list. Applications which require that participants not process arbitrary input messages are also required to perform relevant application-layer input validation checks; see Section 7.7 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, a NonZeroScalar.
- sk_i, Signer secret key share, a Scalar.
- group_public_key, public key corresponding to the group signing key,
  an Element.
- nonce_i, pair of Scalar values (hiding_nonce, binding_nonce) generat
  round one.
- msg, the message to be signed, a byte string.
- 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 a NonZeroScalar identifier j and
  Element values (hiding_nonce_commitment_j, binding_nonce_commitment_
  This list MUST be sorted in ascending order by identifier.

Outputs:
- sig_share, a signature share, a Scalar.

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 the interpolating value
  participant_list = participants_from_commitment_list(commitment_list
  lambda_i = derive_interpolating_value(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 completing sign, and MUST NOT use the nonce as input more than once to sign.

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.

5.3. Signature Share Aggregation

After participants perform round two and send their signature shares to the Coordinator, the Coordinator aggregates each share to produce a final signature. Before aggregating, the Coordinator MUST validate each signature share using DeserializeScalar. If validation fails, the Coordinator MUST abort the protocol as the resulting signature will be invalid. If all signature shares are valid, the Coordinator then aggregates them to produce the final signature using the following procedure.

```
Inputs:
- 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 a NonZeroScalar identifier j and two commitmen
  Element values (hiding_nonce_commitment_j, binding_nonce_commitment_
  This list MUST be sorted in ascending order by identifier.
- msg, the message to be signed, a byte string.
- 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 Scalar z.

def aggregate(commitment_list, msg, sig_shares):
  # Compute the binding factors
  binding_factor_list = compute_binding_factors(commitment_list, msg)

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

  # Compute aggregated signature
  z = Scalar(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). This signature encoding is the same for all FROST ciphersuites specified in Section 6.

The Coordinator SHOULD verify this signature using the group public key before publishing or releasing the signature. Signature verification is as specified for the corresponding ciphersuite; see Section 6 for details. The aggregate signature will verify successfully if all signature shares are valid. Moreover, subsets of valid signature shares will themselves not yield a valid aggregate signature.

If the aggregate signature verification fails, the Coordinator can verify each signature share individually to identify and act on misbehaving participants. The mechanism for acting on a misbehaving participant is out of scope for this specification; see Section 5.4 for more information about dealing with invalid signatures and misbehaving participants.

The function for verifying a signature share, denoted verify_signature_share, is described below. Recall that the Coordinator is configured with "group info" which contains the group public key PK and public keys PK_i for each participant, so the group_public_key and PK_i function arguments should come from that previously stored group info.

```
Inputs:
- identifier, identifier i of the participant, a NonZeroScalar.
- PK_i, the public key for the i-th participant, where PK_i = G.Scalar
  an Element.
- comm_i, pair of Element values in G (hiding_nonce_commitment, bindin
  generated in round one from the i-th participant.
- sig_share_i, a Scalar value indicating the signature share as produc
  round two from the i-th 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 a NonZeroScalar identifier j and two commitmen
  Element values (hiding_nonce_commitment_j, binding_nonce_commitment_
  This list MUST be sorted in ascending order by identifier.
- group_public_key, public key corresponding to the group signing key,
  an Element.
- msg, the message to be signed, a byte string.

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 the interpolating value
  participant_list = participants_from_commitment_list(commitment_list
  lambda_i = derive_interpolating_value(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

 The Coordinator can verify each signature share before first
 aggregating and verifying the signature under the group public key.
 However, since the aggregate signature is valid if all signature
```

shares are valid, this order of operations is more expensive if the signature is valid.

5.4. Identifiable Abort

FROST does not provide robustness; i.e, all participants are required to complete the protocol honestly in order to generate a valid signature. When the signing protocol does not produce a valid signature, the Coordinator SHOULD abort; see Section 7 for more information about FROST's security properties and the threat model.

As a result of this property, a misbehaving participant can cause a denial-of-service on the signing protocol by contributing malformed signature shares or refusing to participate. FROST assumes the network channel is authenticated to identify which signer misbehaved. FROST allows for identifying misbehaving participants that produce invalid signature shares as described in Section 5.3. FROST does not provide accommodations for identifying participants that refuse to participate, though applications are assumed to detect when participants fail to engage in the signing protocol.

In both cases, preventing this type of attack requires the Coordinator to identify misbehaving participants such that applications can take corrective action. The mechanism for acting on misbehaving participants is out of scope for this specification. However, one reasonable approach would be to remove the misbehaving participant from the set of allowed participants in future runs of FROST.

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) as described in Section 6.2. The (Ed25519, SHA-512) and (Ed448, SHAKE256) ciphersuites are included for compatibility with Ed25519 and Ed448 as defined in [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 of 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.

Each ciphersuite adheres to the requirements in Section 6.6. Future ciphersuites MUST also adhere to these requirements.

6.1. FROST(Ed25519, SHA-512)

This ciphersuite uses edwards25519 for the Group and SHA-512 for the Hash function H meant to produce Ed25519-compliant signatures as specified in Section 5.1 of [RFC8032]. The value of the contextString parameter is "FROST-ED25519-SHA512-v11".

* 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. Additionally, this function validates that the input element is not the group identity element.

  - 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. If any of these checks fail, deserialization returns an error. 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. Note that optimizations for this check exist; see [Pornin22].

  - 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, which has 64 bytes of output

  -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 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][z]B = [8]R + [8][c]PK (changed to use the notation in this document).

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-v11".

*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]. Additionally, this function validates that the input element is not the group identity element.

- DeserializeElement(buf): Implemented using the 'Decode' function from [RISTRETTO]. Additionally, this function validates that the resulting element is not the group identity element. If either 'Decode' or that check fails, deserialization returns an error.

- 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, which has 64 bytes of output

- 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 Ed448-compliant signatures as specified in Section 5.2 of [RFC8032]. Note that this ciphersuite does not allow applications to specify a context string as is allowed for Ed448 in [RFC8032], and always sets the [RFC8032]

context string to the empty string. The value of the (internal to FROST) contextString parameter is "FROST-ED448-SHAKE256-v11".

*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. Additionally, this function validates that the input element is not the group identity element.

-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 and is in the prime-order subgroup. If any of these checks fail, deserialization returns an error. 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. Note that optimizations for this check exist; see [Pornin22].

-SerializeScalar(s): Implemented by outputting the little-endian 57-byte encoding of the Scalar value.

-DeserializeScalar(buf): Implemented by attempting to deserialize a Scalar from a little-endian 57-byte string. This function can fail if the input does not represent a Scalar in the range [0, G.Order() - 1].

*Hash (H): SHAKE256 with 114 bytes of output

-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 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][z]B = [4]R + [4][c]PK (changed to use the notation in this document).

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-v11".

*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. Additionally, this function validates that the input element is not the group identity element.

-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 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. (As noted in the specification, validation of the point order is not required since the cofactor is 1.) If any of 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, which has 32 bytes of output

  - 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-v11".

*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], yielding a 33-byte output. Additionally, this function validates that the input element is not the group identity element.

-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 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. (As noted in the specification, validation of the point order is not required since the cofactor is 1.) If any of 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, which has 32 bytes of output

-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.

6.6. Ciphersuite Requirements

Future documents that introduce new ciphersuites MUST adhere to the following requirements.

1. H1, H2, and H3 all have output distributions that are close to (indistinguishable from) the uniform distribution.

2. All hash functions MUST be domain separated with a per-suite context string. Note that the FROST(Ed25519, SHA-512) ciphersuite does not adhere to this requirement for compatibility with [RFC8032].

3. The group MUST be of prime-order, and all deserialization functions MUST output elements that belong to their respective sets of Elements or Scalars, or failure when deserialization fails.

7. Security Considerations

A security analysis of FROST exists in [FROST20] and [StrongerSec22]. At a high level, FROST provides security against Existential Unforgeability Under Chosen Message Attack (EUF-CMA) attacks, as defined in [StrongerSec22]. Satisfying this requirement

requires the ciphersuite to adhere to the requirements in Section 6.6, as well as the following assumptions to hold.

- The signer key shares are generated and distributed securely, e.g., via a trusted dealer that performs key generation (see Appendix C.2) or through a distributed key generation protocol.

- The Coordinator and at most (MIN_PARTICIPANTS-1) participants may be corrupted.

Note that the Coordinator is not trusted with any private information and communication at the time of signing can be performed over a public channel, as long as it is authenticated and reliable.

FROST provides security against denial of service attacks under the following assumptions:

- The Coordinator does not perform a denial of service attack.

- The Coordinator identifies misbehaving participants such that they can be removed from future invocations of FROST. The Coordinator may also abort upon detecting a misbehaving participant to ensure that invalid signatures are not produced.

FROST does not aim to achieve the following goals:

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

- Robustness. Preventing denial-of-service attacks against misbehaving participants requires the Coordinator to identify and act on misbehaving participants; see Section 5.4 for more information. While FROST does not provide robustness, [ROAST] is as a wrapper protocol around FROST that does.

- 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. Side-channel mitigations

Several routines process secret values (nonces, signing keys / shares), and depending on the implementation and deployment

environment, mitigating side-channels may be pertinent. Mitigating these side-channels requires implementing G.ScalarMult(), G.ScalarBaseMult(), G.SerializeScalar(), and G.DeserializeScalar() in constant (value-independent) time. The various ciphersuites lend themselves differently to specific implementation techniques and ease of achieving side-channel resistance, though ultimately avoiding value-dependent computation or branching is the goal.

7.2. Optimizations

[StrongerSec22] presented an optimization to FROST that reduces the total number of scalar multiplications from linear in the number of signing participants to a constant. However, as described in [StrongerSec22], 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. As such, the optimization is NOT RECOMENDED, and it is not covered in this document.

7.3. Nonce Reuse Attacks

Section 4.1 describes the procedure that participants use to produce nonces during the first round of signing. 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.

7.4. 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.

7.5. 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.

7.6. 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 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. For example, if a fictional protocol Quux decided to pre-hash its input messages, one possible way to do so is via H(contextString || "Quux-pre-hash" || m).

7.7. Input Message Validation

Message validation varies by application. For example, some applications may require that participants only process messages of a certain structure. 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, some applications may require that participants only process messages with permitted content according to some

policy. In digital currency applications, this might mean that a transaction being signed is allowed and intended by the relevant stakeholders. Another instance of this type of message validation is in the context of [TLS], wherein implementations may use threshold signing protocols to produce signatures of transcript hashes. In this setting, signing participants might require the raw TLS handshake messages to validate before computing the transcript hash that is signed.

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. References

8.1. Normative References

[HASH-TO-CURVE] Faz-Hernandez, 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>.

[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-decaf448-05, 29 November 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-05>.

[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.

8.2. Informative References

[FeldmanSecretSharing] Feldman, P., "A practical scheme for non-interactive verifiable secret sharing", 28th Annual Symposium on Foundations of Computer Science (sfcs 1987), DOI 10.1109/sfcs.1987.4, October 1987, <https://doi.org/10.1109/sfcs.1987.4>.

[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>.

[ROAST] Ruffing, T., Ronge, V., Jin, E., Schneider-Bensch, J., and D. Schröder, "ROAST: Robust Asynchronous Schnorr Threshold Signatures", 18 September 2022, <https://eprint.iacr.org/2022/550>.

[ShamirSecretSharing] Shamir, A., "How to share a secret", Communications of the ACM vol. 22, no. 11, pp. 612-613, DOI 10.1145/359168.359176, November 1979, <https://doi.org/10.1145/359168.359176>.

[StrongerSec22] Bellare, M., Crites, E., Komlo, C., Maller, M., Tessaro, S., and C. Zhu, "Better than Advertised Security for Non-interactive Threshold Signatures", 1 June 2022, <https://crypto.iacr.org/2022/papers/538806_1_En_18_Chapter_OnlinePDF.pdf>.

[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 Scalar z.

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:
- True if signature is valid, and False 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 = 1, ..., 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_PARTICIPANTS 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 (a NonZeroScalar) and the k
- group_public_key, public key corresponding to the group signing key,
  Element.
- 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. The dealer can use a secure broadcast channel to ensure each participant has a consistent view of this 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 at least 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. The algorithm polynomial_evaluate is defined in Appendix C.1.1.

```
secret_share_shard(s, coefficients, MAX_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

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

def secret_share_shard(s, coefficients, MAX_PARTICIPANTS):
  # 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_shares.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_interpolate_constant is defined in Appendix C.1.1.

```
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:
- s, the resulting secret that was previously split into shares, a Sca

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_interpolate_constant(shares)
  return s
```

C.1.1. Additional polynomial operations

This section describes two functions. One function, denoted polynomial_evaluate, is for evaluating a polynomial f(x) at a particular point x using Horner's method, i.e., computing y = f(x). The other function, polynomial_interpolate_constant, is for recovering the constant term of an interpolating polynomial defined by a set of points.

The function polynomial_evaluate is defined as follows.

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

The function polynomial_interpolate_constant is defined as follows.

```
Inputs:
- points, a set of t points with distinct x coordinates on a polynomia
  each a tuple of two Scalar values representing the x and y coordinat

Outputs:
- f_zero, the constant term of f, i.e., f(0), a Scalar.

def polynomial_interpolate_constant(points):
  x_coords = []
  for (x, y) in points:
    x_coords.append(x)

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

  return f_zero
```

C.2. Verifiable Secret Sharing

Feldman's Verifiable Secret Sharing (VSS) [FeldmanSecretSharing] 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:
- vss_commitment, a vector commitment to each of the coefficients in c
each item of the vector commitment is an Element.

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 (a NonZeroScalar), and sk_i the participant's secret key,
  secret share of the 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:
- True if sk_i is valid, and False otherwise.

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

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 int
- vss_commitment, a VSS commitment to a secret polynomial f, a vector
coefficients in coeffs, where each element of the vector commitment is

Outputs:
- PK, the public key representing the group, an Element.
- participant_public_keys, a list of MAX_PARTICIPANTS public keys PK_i
  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 DeserializeScalar in constant time can leak information about the underlying corresponding Scalar.

As an optimization, if the group order is very close to a power of 2, it is acceptable to omit the rejection test completely. In particular, if the group order is p, and there is an integer b such that $|p - 2^b|$ is less than $2^{(b/2)}$, then RandomScalar can simply return a uniformly random integer of at most b bits.

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: 9d06a6381c7a4493929761a73692776772b274236
fb5cfcc7d1b48ac3a9c249f
P1 binding_nonce_randomness: db184d7bc01a3417fe1f2eb3cf5479bb027145e6
369a5f879f32d334ab256b23
P1 hiding_nonce: 70652da3e8d7533a0e4b9e9104f01b48c396b5b553717784ed8d
05c6a36b9609
P1 binding_nonce: 4f9e1ad260b5c0e4fe0e0719c6324f89fecd053758f77c957f5
6967e634a710e
P1 hiding_nonce_commitment: 44105304351ceddc58e15ddea35b2cb48e60ced54
ceb22c3b0e5d42d098aa1d8
P1 binding_nonce_commitment: b8274b18a12f2cef74ae42f876cec1e31daab5cb
162f95a56cd2487409c9d1dd
P1 binding_factor_input: c5b95020cba31a9035835f074f718d0c3af02a318d6b
4723bbd1c088f4889dd7b9ff8e79f9a67a9d27605144259a7af18b7cca2539ffa5c4f
1366a98645da8f4e077d604fff64f20e2377a37e5a10ce152194d62fe856ef4cd935d
4f1cb0088c2083a2722ad3f5a84d778e257da0df2a7cadb004b1f5528352af778b94e
e1c2a0100000000000000000000000000000000000000000000000000000000000000
P1 binding_factor: 2d5630c36d33258b1208c4205fa759b762d09bfa06b29cf792
cf98758c0b3305
P3 hiding_nonce_randomness: 31ca9b07936d6b342a43d97f23b7bec5a5f5a0957
5a075393868dd8df5c05a54
P3 binding_nonce_randomness: c1db96a85d8b593e14fdb869c0955625478afa6a
987ad217e7f2261dcab26819
```

P3 hiding_nonce: 233adcb0ec0eddba5f1cc5268f3f4e6fc1dd97fb1e4a1754e6dd
c92ed834ca0b
P3 binding_nonce: b59fc8a32fe02ec0a44c4671f3d1f82ea3924b7c7c0179398fc
9137e82757803
P3 hiding_nonce_commitment: d31bd81ce216b1c83912803a574a0285796275cb8
b14f6dc92c8b09a6951f0a2
P3 binding_nonce_commitment: e1c863cfd08df775b6747ef2456e9bf9a03cc281
a479a95261dc39137fcf0967
P3 binding_factor_input: c5b95020cba31a9035835f074f718d0c3af02a318d6b
4723bbd1c088f4889dd7b9ff8e79f9a67a9d27605144259a7af18b7cca2539ffa5c4f
1366a98645da8f4e077d604fff64f20e2377a37e5a10ce152194d62fe856ef4cd935d
4f1cb0088c2083a2722ad3f5a84d778e257da0df2a7cadb004b1f5528352af778b94e
e1c2a03000000000000000000000000000000000000000000000000000000000000
P3 binding_factor: 1137be5cdf3d18e44367acee8485e9a66c3164077af80619b6
291e3943bbef04

// Round two parameters
participant_list: 1,3

// Signer round two outputs
P1 sig_share: c4b26af1e91fbc8440a0dad253e72620da624553c5b625fd51e6ea1
79fc09f05
P3 sig_share: 9369640967d0cb98f4dedfde58a845e0e18e0a7164396358439060e
d282b4e08

sig: ae11c539fdc709b78fef5ee1f5a2250297e3e1b62a86a86c26d93c389934ba0e
571ccffa50f0871d357fbab1ac8f6c00bcf14fc429f0885595764b05c8ebed0d

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: 89bf16040081ff2990336b200613787937ebe1f02
4b8cdff90eb6f1c741d91c1
P1 binding_nonce_randomness: cd646348bb98fd2a4b2f27fb7d6da18201c16184
7352576b4bf125190e965483
P1 hiding_nonce: 67a6f023e77361707c6e894c625e809e80f33fdb310810053ae2
9e28e7011f3193b9020e73c183a98cc3a519160ed759376dd92c9483162200
P1 binding_nonce: 4812e8d7c8b7a50ced80b507902d074ef8647bc1146979683da
8d0fecd93fa3c8230cade2fb4344600aa04bd4b7a21d046c5b63ee865b12a00
P1 hiding_nonce_commitment: 649c6a53b109897d962d033f23d01fd4e1053dddf
3746d2ddce9bd66aea38ccfc3df061df03ca399eb806312ab3037c0c31523142956ad
a780
P1 binding_nonce_commitment: 0064cc729a8e2fcf417e43788ecec37b10e9e1dc
b3ae90854efbfaad00a0ef3cdd52e18d56f073c8ff0947cb71ff0bb17c3d45d096409
ddb00
P1 binding_factor_input: 106dadce87ca867018702d69a02effd165e1ac1a511c
957cff1897ceff2e34ca212fe798d84f0bde6054bf0fa77fd4cd4bc4853d6dc8dbd19
d340923f0ebbbb35172df4ab865a45d55af31fa0e6606ea97cf8513022b2b133d0f9f
6b8d3be184221fc4592bf12bd7fb4127bb67e51a6dc9e5f1ed5243362fb46a6da5524
18ca967d43d9bc811a21917a3018de58f11c25f6b9ad8bec3699e06b87dd3ab67a732
6c30878c7c55ec1a45802af65da193ce99634158539e38c232a627895c5f14e2e20d4
87382ccc9c99cd0a0df266a292f283bb9b6854e344ecc32d5e1852fdde5fde7779801
0000000000000000000000000000000000000000000000000000000000000000000
```

```
0000000000000000000000000000000000000000000000000
P1 binding_factor: 3412ac894a91a6bc0e3e7c790f3e8ef5d1288e54de780aba38
4cbb3081b602dd188010e5b0c9ac2b5dca0aae54cfd0f5c391cece8092131d00
P3 hiding_nonce_randomness: 3718dabb4fd3d7dd9adad4878c6de8b33c8841cfe
7cc95a85592952a2c9c554d
P3 binding_nonce_randomness: 3becbc90798211a0f52543dd1f24869a143fdf74
3409581af4db30f045773d64
P3 hiding_nonce: 4f2666770317d14ec9f7fd6690c075c34b4cde7f6d9bceda9e94
33ec8c0f2dc983ff1622c3a54916ce7c161381d263fad62539cddab2101600
P3 binding_nonce: 88f66df8bb66389932721a40de4aa5754f632cac114abc10526
88104d19f3b1a010880ebcd0c4c0f8cf567d887e5b0c3c0dc78821166550f00
P3 hiding_nonce_commitment: 8dcf049167e28d5f53fa7ebbbd136abcaf2be9f2c
02448c8979002f92577b22027640def7ddd5b98f9540c2280f36a92d4747bbade0b0c
4280
P3 binding_nonce_commitment: 12e837b89a2c085481fcf0ca640a17a24b6fc96b
032d40e4301c78e7232a9f49ffdcad2c21acbc992e79dfc3c6c07cb94e4680b3dcc99
35580
P3 binding_factor_input: 106dadce87ca867018702d69a02effd165e1ac1a511c
957cff1897ceff2e34ca212fe798d84f0bde6054bf0fa77fd4cd4bc4853d6dc8dbd19
d340923f0ebbbb35172df4ab865a45d55af31fa0e6606ea97cf8513022b2b133d0f9f
6b8d3be184221fc4592bf12bd7fb4127bb67e51a6dc9e5f1ed5243362fb46a6da5524
18ca967d43d9bc811a21917a3018de58f11c25f6b9ad8bec3699e06b87dd3ab67a732
6c30878c7c55ec1a45802af65da193ce99634158539e38c232a627895c5f14e2e20d4
87382ccc9c99cd0a0df266a292f283bb9b6854e344ecc32d5e1852fdde5fde7779803
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000
P3 binding_factor: 6aa48a3635d7b962489283ee1ccda8ea66e5677b1e17f2f475
eb565e3ae8ea73360f24c04e3775dadd1f2923adcda3d105536ad28c3c561100

// Round two parameters
participant_list: 1,3

// Signer round two outputs
P1 sig_share: c5057c80d13e565545dac6f3aa333065c809a14a94fea3c8e4e87e3
86a9cb89602de7355c5d19ebb09d553b100ef1858104fc7c43992d83400
P3 sig_share: 2b490ea08411f78c620c668fff8ba70b25b7c89436f20cc45419213
de70f93fb6c9094c79293697d72e741b68d2e493446005145d0b7fc3500

sig: 83ac141d289a5171bc894b058aee2890316280719a870fc5c1608b7740302315
5d7a9dc15a2b7920bb5826dd540bf76336be99536cebe36280fd093275c38dd4be525
767f537fd6a4f5d8a9330811562c84fded5f851ac4b926f6e081d586508397cbc9567
8e1d628c564f180a0a4ad52a00
```

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: 81800157bb554f299fe0b6bd658e4c4591d74168b
5177bf55e8dceed59dc80c7
P1 binding_nonce_randomness: e9b37de02fde28f601f09051ed9a277b02ac81c8
03a5c72492d58635001fe355
P1 hiding_nonce: 40f58e8df202b21c94f826e76e4647efdb0ea3ca7ae7e3689bc0
cbe2e2f6660c
P1 binding_nonce: 373dd42b5fe80e88edddf82e03744b6a12d59256f546de612d4
bbd91a6b1df06
P1 hiding_nonce_commitment: b8c7319a56b296537436e5a6f509a871a3c74eff1
534ec1e2f539ccd8b322411
P1 binding_nonce_commitment: 7af5d4bece8763ce3630370adbd978699402f624
fd3a7d2c71ea5839efc3cf54
P1 binding_factor_input: 9c245d5fc2e451c5c5a617cc6f2a20629fb317d9b1c1
915ab4bfa319d4ebf922c54dd1a5b3b754550c72734ac9255db8107a2b01f361754d9
f13f428c2f6de9e4f609ae0dbe8bd1f95bee9f9ea219154d567ef174390bac737bb67
ee1787c8a34279728d4aa99a6de2d5ce6deb86afe6bc68178f01223bb5eb934c8a23b
6354e0100000000000000000000000000000000000000000000000000000000000000
P1 binding_factor: 607df5e2e3a8b5e2704716693e18f548100a32b86a5685d393
2a774c3f107e06
P3 hiding_nonce_randomness: daeb223c4a913943cff2fb0b0e638dfcc281e1e89
36ee6c3fef4d49ad9cbfaa0
P3 binding_nonce_randomness: c425768d952ab8f18b9720c54b93e612ba2cca17
0bb7518cac080896efa7429b
```

P3 hiding_nonce: 491477c9dbe8717c77c6c1e2c5f4cec636c7c154313a44c91fea
63e309f3e100
P3 binding_nonce: 3ae1bba7d6f2076f81596912dd916efae5b3c2ef89695632119
4fdd2e52ebc0f
P3 hiding_nonce_commitment: e4466b7670ac4f9d9b7b67655860dd1ab341be18a
654bb1966df53c76c85d511
P3 binding_nonce_commitment: ce47cd595d25d7effc3c095efa2a687a1728a5ec
ab402b39e0c0ad9a525ea54f
P3 binding_factor_input: 9c245d5fc2e451c5c5a617cc6f2a20629fb317d9b1c1
915ab4bfa319d4ebf922c54dd1a5b3b754550c72734ac9255db8107a2b01f361754d9
f13f428c2f6de9e4f609ae0dbe8bd1f95bee9f9ea219154d567ef174390bac737bb67
ee1787c8a34279728d4aa99a6de2d5ce6deb86afe6bc68178f01223bb5eb934c8a23b
6354e0300000000000000000000000000000000000000000000000000000000000
P3 binding_factor: 2bd27271c28746eb93e2114d6778c12b44c9287d84b85dc780
eb08da6f689900

// Round two parameters
participant_list: 1,3

// Signer round two outputs
P1 sig_share: c38f438c325ce6bfa4272b37e7707caaeb57fa8c7ddcc05e0725acb
8a7d9cd0c
P3 sig_share: 4cb9917be3bd53f1d60f1c3d1a3ff563565fa15a391133e7f980e55
d3aeb7904

sig: 204d5d93aa486192ecf2f64ce7dbc1db76948fb1077d1a719ae1ecca6143501e
2275dfaafbb62759a59a4fd122b692f941b79be7b6edf34501a69116e2c44701

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: f4e8cf80aec3f888d997900ac7e3e349944b5a6b4
7649fc32186d2f1238103c6
P1 binding_nonce_randomness: a7f220770b6f10ff54ec6afa55f99bd08cc92fa1
a488c86e9bf493e9cb894cdf
P1 hiding_nonce: f871dfcf6bcd199342651adc361b92c941cb6a0d8c8c1a3b91d7
9e2c1bf3722d
P1 binding_nonce: bd3ece3634a1b303dea0586ed67a91fe68510f11ebe66e88683
09b1551ef2388
P1 hiding_nonce_commitment: 03987febbc67a8ed735affdff4d3a5adf22c05c80
f97f311ab7437a3027372deb3
P1 binding_nonce_commitment: 02a1960477d139035b986d6adcb06491378beb92
ccd097ad94e76291c52343849d
P1 binding_factor_input: 350c8b523feea9bb35720e9fbe0405ed48d78caa4fb6
0869f34367e144c68bb0fc77bf512409ad8b91e2ace4909229891a446c45683f5eb2f
843dbec224527dc0000000000000000000000000000000000000000000000000000
0000000001
P1 binding_factor: cb415dd1d866493ee7d2db7cb33929d7e430e84d80c58070e2
bbb1fdbf76a9c8
P3 hiding_nonce_randomness: 1b6149d252a0a0a6618b8d22a1c49897f9b0d23a4
8f19598e191e05dc7b7ae33
P3 binding_nonce_randomness: e13994bb75aafe337c32afdbfd08ae60dd108fc7
68845edaa871992044cabf1b
P3 hiding_nonce: 802e9321f9f63688c6c1a9681a4a4661f71770e0cef92b8a5997
```

155d18fb82ef
P3 binding_nonce: 8b6b692ae634a24536f45dda95b2398af71cd605fb7a0bbdd94
08d211ab99eba
P3 hiding_nonce_commitment: 0212cac45ebd4100c97506939391f9be4ffc3ca29
60e2ef95aeaa38abdede204ca
P3 binding_nonce_commitment: 03017ce754d310eabda0f5681e61ce3d713cdd33
7070faa6a68471af49694a4e7e
P3 binding_factor_input: 350c8b523feea9bb35720e9fbe0405ed48d78caa4fb6
0869f34367e144c68bb0fc77bf512409ad8b91e2ace4909229891a446c45683f5eb2f
843dbec224527dc0000000000000000000000000000000000000000000000000000
0000000003
P3 binding_factor: dfd82467569334e952edecb10d92adf85b8e299db0b40be313
1a12efdfa3e796

// Round two parameters
participant_list: 1,3

// Signer round two outputs
P1 sig_share: c5acd980310aaf87cb7a9a90428698ef3e6b1e5860f7fb06329bc0e
fe3f14ca5
P3 sig_share: 1e064fbd35467377eb3fe161ff975e9ec3ed8e2e0d4c73f3a6b0a02
3777e1264

sig: 029e07d4171dbf9a730ed95e9d95bda06fa4db76c88c519f7f3ca5483019f46c
b0e3b3293d665122ffb6ba7bf2421df78e0258ac866e446ef9d94c61135b6f5f09

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: 80cbea5e405d169999d8c4b30b755fedb26ab07ec
8198cda4873ed8ce5e16773
P1 binding_nonce_randomness: f6d5b38197843046b68903048c1feba433e35001
45281fa8bb1e26fdfeef5e7f
P1 hiding_nonce: acc83278035223c1ba464e2d11bfacfc872b2b23e1041cf5f613
0da21e4d8068
P1 binding_nonce: c3ef169995bc3d2c2d48f30b83d0c63751e67ceb057695bcb2a
6aa40ed5d926b
P1 hiding_nonce_commitment: 036673d68a928793c33ae07776908eae8ea15dd94
7ed81284e939aaba118573a5e
P1 binding_nonce_commitment: 03d2a96dd4ec1ee29dc22067109d1290dabd8016
cb41856ee8ff9281c3fa1baffd
P1 binding_factor_input: a645d8249457bbcac34fa7b740f66bcce08fc39506b8
bbf1a1c81092f6272eda82ae39234d714f87a7b91dd67d124a06561a36817c1ecaa25
5c3527d694fc4f1000000000000000000000000000000000000000000000000000
0000000001
P1 binding_factor: d7bcbd29408dedc9e138262d99b09d8b5705d76eb5de2369d9
103e4423f8ac79
P3 hiding_nonce_randomness: b9794047604beda0c5c0529ac9dfd83c0a80399a7
bdf4c3e23cef2faf69cdcc3
P3 binding_nonce_randomness: c28ce6252631620b84c2702b34774fab365e286e
bc77030a112ebccccbffa78b
P3 hiding_nonce: cb3387defef07fc9010c0564ba6495ed41876626ed86b886ca26
```

```
cbbd3566ffbc
P3 binding_nonce: 4559459735eb68e8c16319a9fd9a14016053957cb8cea273a24
b7c7bc1ee26f6
P3 hiding_nonce_commitment: 030278e6e6055fb963b40e0c3c37099f803f3f389
30fc89092517f8ce1b47e8d6b
P3 binding_nonce_commitment: 028eb6d238c6c0fc6216906706ad0ff9943c6c1d
6079cdf74f674481ebb2485db3
P3 binding_factor_input: a645d8249457bbcac34fa7b740f66bcce08fc39506b8
bbf1a1c81092f6272eda82ae39234d714f87a7b91dd67d124a06561a36817c1ecaa25
5c3527d694fc4f1000000000000000000000000000000000000000000000000000
0000000003
P3 binding_factor: ecc057259f3c8b195308c9b73aaaf840660a37eb264ebce342
412c58102ee437

// Round two parameters
participant_list: 1,3

// Signer round two outputs
P1 sig_share: 1750b2a314a81b66fd81366583617aaafcffa68f14495204795aa04
34b907aa3
P3 sig_share: e4dbbbbbcb035eb3512918b0368c4ab2c836a92dccff3251efa7a4a
acc7d3790

sig: 0259696aac722558e8638485d252bb2556f6241a7adfdf284c8c87a3428d4644
8dfc2c6e5edfab7a1a4eaa4f15b9edc55dc5364fbce1488456690244ee180db233
```

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