Internet Research Task Force (IRTF)
Request for Comments: 9591
Category: Informational
ISSN: 2070-1721

D. Connolly
Zcash Foundation
C. Komlo
University of Waterloo, Zcash Foundation
I. Goldberg
University of Waterloo
C. A. Wood
Cloudflare

June 2024

# The Flexible Round-Optimized Schnorr Threshold (FROST) Protocol for Two-Round Schnorr Signatures

## 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. This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

## Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Research Task Force (IRTF). The IRTF publishes the results of Internet-related research and development activities. These results might not be suitable for deployment. This RFC represents the consensus of the Crypto Forum Research Group of the Internet Research Task Force (IRTF). Documents approved for publication by the IRSG are not candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9591.



## Table of Contents

## 1. Introduction

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 a verifier that is compliant with [RFC8032]. 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.

## 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.

byte: A sequence of eight bits.

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 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)

The following sections describe these constructs in more detail.

### 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, 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 that 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 the ciphersuite used. 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);

* List operations (Section 4.3);

* Binding factors 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 random number generator (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 as 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.

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 three 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 that is used for polynomial interpolation. It is provided a list of x-coordinates as input, each of which cannot equal 0.

Inputs:
- L, the list of x-coordinates, each a NonZeroScalar.
- x_i, an x-coordinate contained in L, 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(L, x_i):
  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_commitment_i), ...], a list of commitments issued by each participant, where each element in the list indicates a NonZeroScalar identifier i and two commitment Element values (hiding_nonce_commitment_i, binding_nonce_commitment_i). This list MUST be sorted in ascending order by identifier.

Outputs:
- encoded_group_commitment, the serialized representation of commitment_list, a byte string.

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

```
        G.SerializeElement(binding_nonce_commitment))
    encoded_group_commitment = (
        encoded_group_commitment ||
        encoded_commitment)
  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_commitment_i), ...], a list of commitments issued by each participant, where each element in the list indicates a NonZeroScalar identifier i and two commitment Element values (hiding_nonce_commitment_i, binding_nonce_commitment_i). This list MUST be sorted 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 in the list 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, message to be signed, and group public key.

Inputs:
- group_public_key, the public key corresponding to the group signing key, an Element.
- commitment_list = [(i, hiding_nonce_commitment_i, binding_nonce_commitment_i), ...], a list of commitments issued by each participant, where each element in the list indicates a NonZeroScalar identifier i and two commitment Element values (hiding_nonce_commitment_i, binding_nonce_commitment_i). This list MUST be sorted in ascending order by identifier.
- msg, the message to be signed.

```
Outputs:
- binding_factor_list, a list of (NonZeroScalar, Scalar) tuples
  representing the binding factors.

def compute_binding_factors(group_public_key, commitment_list, msg):
  group_public_key_enc = G.SerializeElement(group_public_key)
  // Hashed to a fixed length.
  msg_hash = H4(msg)
  // Hashed to a fixed length.
  encoded_commitment_hash =
      H5(encode_group_commitment_list(commitment_list))
  // The encoding of the group public key is a fixed length
  // within a ciphersuite.
  rho_input_prefix = group_public_key_enc || msg_hash ||
   encoded_commitment_hash

  binding_factor_list = []
  for (identifier, hiding_nonce_commitment,
       binding_nonce_commitment) in commitment_list:
    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), ...], a list of commitments issued by
  each participant, where each element in the list indicates a
  NonZeroScalar identifier i and two commitment Element values
  (hiding_nonce_commitment_i, binding_nonce_commitment_i). This list
  MUST be sorted in ascending order by identifier.
- binding_factor_list = [(i, binding_factor), ...],
  a list of (NonZeroScalar, Scalar) tuples representing the binding
  factor Scalar 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) in commitment_list:
    binding_factor = binding_factor_for_participant(
        binding_factor_list, identifier)
    binding_nonce = G.ScalarMult(
        binding_nonce_commitment,
        binding_factor)
    group_commitment = (
        group_commitment +
        hiding_nonce_commitment +
        binding_nonce)
  return group_commitment
```

Note that the performance of this algorithm is defined naively and scales linearly relative to the number of signers. For improved performance, the group commitment can be computed using multi-exponentiation techniques such as Pippinger’s algorithm; see [MultExp] for more details.

## 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 key, an 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 and non-zero integer that MUST be at least MIN_PARTICIPANTS, but MUST NOT be larger than MAX_PARTICIPANTS, where MIN_PARTICIPANTS <= MAX_PARTICIPANTS and MIN_PARTICIPANTS is a positive and non-zero integer. Additionally, MAX_PARTICIPANTS MUST be 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 the participants that will participate (at least MIN_PARTICIPANTS in number);

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

3. Aggregating signature shares output by each participant; and

4. 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 designating a single 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 is used to produce signatures; see Appendix C.1 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 as 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).

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

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

* Public keys PK_i for each participant, which are Element values in G denoted as 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 configuration mechanisms are possible: one that requires a single trusted dealer and one that requires performing a distributed key generation protocol. We highlight the 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 is completed, the Coordinator aggregates the signature shares to produce a final signature. The Coordinator SHOULD abort the protocol if the signature is invalid; see Section 5.4 for more information about dealing with invalid signatures and misbehaving participants. This complete interaction (without being aborted) is shown in Figure 1.

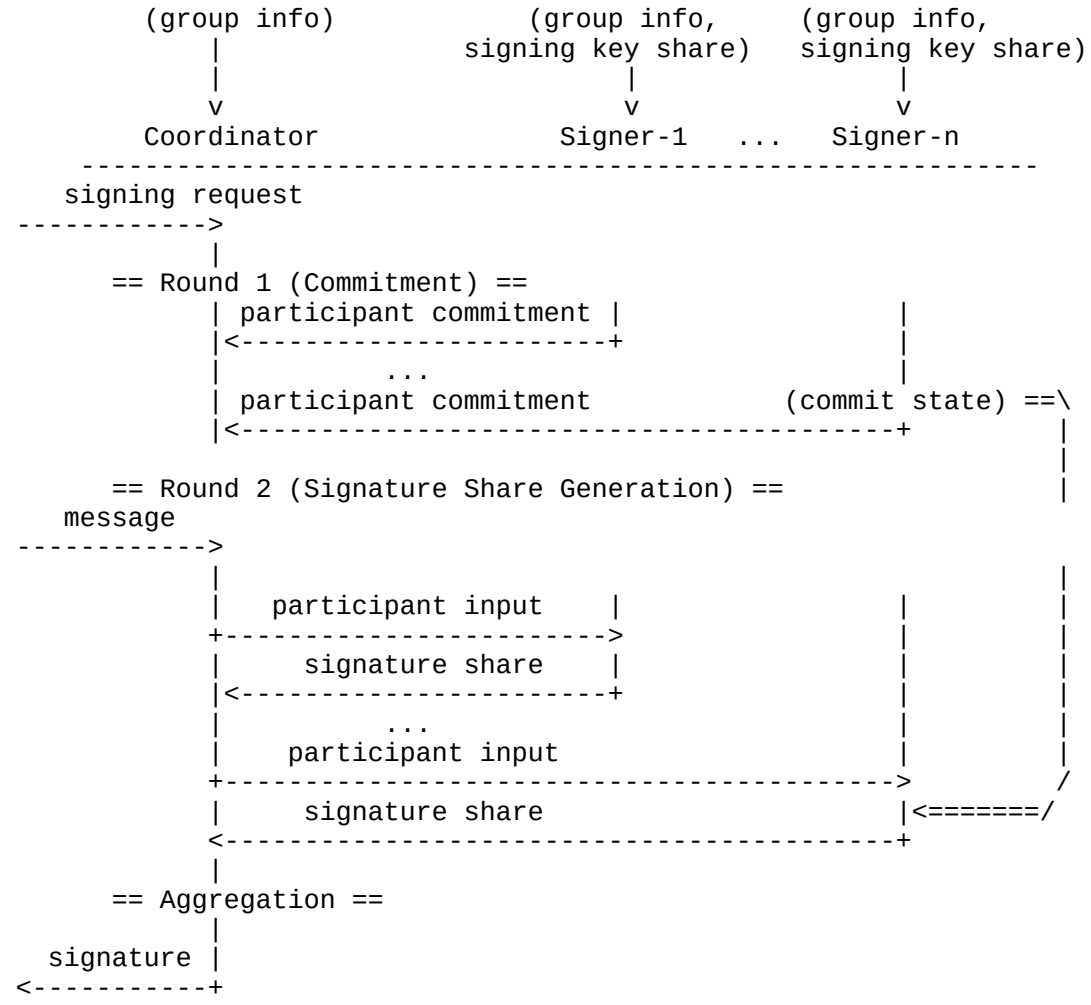


Figure 1: FROST Protocol Overview

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; 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 that are received over the network, are validated before use. In particular, this means that any value of type Element or Scalar received over the network MUST be deserialized using DeserializeElement and DeserializeScalar, respectively, as these functions perform the necessary input validation steps. Additionally, all messages sent over the wire MUST be encoded using their respective functions, e.g., Scalars and Elements are encoded using SerializeScalar and SerializeElement.

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)
  nonces = (hiding_nonce, binding_nonce)
  comms = (hiding_nonce_commitment, binding_nonce_commitment)
  return (nonces, comms)
```

The outputs nonce and comm from participant P_i are both 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 choosing the participants that will participate (a number of 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 that restrict participants from processing 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) generated in round one.
- msg, the message to be signed, a byte string.
- commitment_list = [(i, hiding_nonce_commitment_i, binding_nonce_commitment_i), ...], a list of commitments issued by each participant, where each element in the list indicates a NonZeroScalar identifier i and two commitment Element values (hiding_nonce_commitment_i, binding_nonce_commitment_i). 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_list):
  # Compute the binding factor(s)
  binding_factor_list = compute_binding_factors(group_public_key,
   commitment_list, msg)
  binding_factor = binding_factor_for_participant(
      binding_factor_list, identifier)

  # Compute the group commitment
  group_commitment = compute_group_commitment(
      commitment_list, binding_factor_list)

  # Compute the interpolating value
  participant_list = participants_from_commitment_list(
      commitment_list)
  lambda_i = derive_interpolating_value(participant_list, identifier)

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

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

  return sig_share
```

The output of this procedure is a signature share. Each participant

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 aggregates them to produce the final signature using the following procedure.

Inputs:
- commitment_list = [(i, hiding_nonce_commitment_i, binding_nonce_commitment_i), ...], a list of commitments issued by each participant, where each element in the list indicates a NonZeroScalar identifier i and two commitment Element values (hiding_nonce_commitment_i, binding_nonce_commitment_i). This list MUST be sorted in ascending order by identifier.
- msg, the message to be signed, a byte string.
- group_public_key, public key corresponding to the group signing key, an Element.
- sig_shares, a set of signature shares z_i, Scalar values, for each participant, of length NUM_PARTICIPANTS, where MIN_PARTICIPANTS <= NUM_PARTICIPANTS <= MAX_PARTICIPANTS.

Outputs:
- (R, z), a Schnorr signature consisting of an Element R and Scalar z.

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

  # Compute the group commitment
  group_commitment = compute_group_commitment(
      commitment_list, binding_factor_list)

  # Compute aggregated signature
  z = Scalar(0)
  for z_i in sig_shares:
    z = z + z_i
  return (group_commitment, z)
```

The output from the aggregation step is the output signature (R, z). The canonical encoding of this signature is 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 not yield a valid aggregate signature themselves.

If the aggregate signature verification fails, the Coordinator MAY 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 as verify_signature_share, is described below. Recall that the Coordinator is configured with "group info" that contains the group public key PK and public keys PK_i for each participant. The group_public_key and PK_i function arguments MUST 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.ScalarBaseMult(sk_i), an Element.
- comm_i, pair of Element values in G (hiding_nonce_commitment, binding_nonce_commitment) generated in round one from the i-th participant.
- sig_share_i, a Scalar value indicating the signature share as produced in round two from the i-th participant.
- commitment_list = [(i, hiding_nonce_commitment_i, binding_nonce_commitment_i), ...], a list of commitments issued by each participant, where each element in the list indicates a NonZeroScalar identifier i and two commitment Element values (hiding_nonce_commitment_i, binding_nonce_commitment_i). 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, commitment_list,
        group_public_key, msg):
  # Compute the binding factors
  binding_factor_list = compute_binding_factors(group_public_key,
   commitment_list, msg)
  binding_factor = binding_factor_for_participant(
      binding_factor_list, identifier)

  # Compute the group commitment
  group_commitment = compute_group_commitment(
      commitment_list, binding_factor_list)

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

  # Compute the challenge
  challenge = compute_challenge(
      group_commitment, group_public_key, msg)

  # Compute the interpolating value
  participant_list = participants_from_commitment_list(
      commitment_list)
  lambda_i = derive_interpolating_value(participant_list, identifier)

  # 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 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 (DoS) on the signing protocol by contributing malformed signature shares or refusing to participate. Identifying misbehaving participants that produce invalid shares can be done by checking signature shares from each participant using verify_signature_share as described in Section 5.3. FROST assumes the network channel is authenticated to identify the signer that 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 as contextString, which is an ASCII string literal (with no terminating NUL 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-v1".

Group: edwards25519 [RFC8032], where Ne = 32 and Ns = 32.

- 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 an output of 64 bytes.

- 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; however, 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).

Canonical signature encoding is as specified in Appendix A.

### 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-v1".

Group: ristretto255 [RISTRETTO], where Ne = 32 and Ns = 32.

- 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 the "Decode" function or the 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.

Canonical signature encoding is as specified in Appendix A.

### 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]. Unlike Ed448 in [RFC8032], this ciphersuite does not allow applications to specify a context string and always sets the context of [RFC8032] to the empty string. 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-v1".

Group: edwards448 [RFC8032], where Ne = 57 and Ns = 57.

- Order(): Return 2^446 - 13818066809895115352007386748515426880336692474882178609894547503885.
- 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 - 1381806680989511535200738674851542688033669247488217860989454750388 5.
- 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 - 1381806680989511535200738674851542688033669247488217860989454750388 5.
- 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 - 1381806680989511535200738674851542688033669247488217860989454750388 5.
- 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. However, it is omitted for compatibility with [RFC8032].

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).

Canonical signature encoding is as specified in Appendix A.

### 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-v1".

Group: P-256 (secp256r1) [x9.62], where Ne = 33 and Ns = 32.

- 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 performing 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) (see [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) (see [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) (see [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.

Canonical signature encoding is as specified in Appendix A.

### 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-v1".

Group: secp256k1 [SEC2], where Ne = 33 and Ns = 32.

Order(): Return 0xfffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141.

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 performing 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, the point is on the curve, and 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) (see [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) (see [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) (see [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.

Canonical signature encoding is as specified in Appendix A.

### 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 H2 alone in order to maintain 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 else fail.

4. The canonical signature encoding details are clearly specified.

## 7. Security Considerations

A security analysis of FROST is documented in [FROST20] and [StrongerSec22]. At a high level, FROST provides security against Existential Unforgeability Under Chosen Message Attacks (EUF-CMA) as defined in [StrongerSec22]. To satisfy this requirement, the ciphersuite needs to adhere to the requirements in Section 6.6 and the following assumptions must 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 DoS attacks under the following assumptions:

* The Coordinator does not perform a DoS 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 DoS 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 a wrapper protocol around FROST that does.

* Downgrade prevention. All participants in the protocol are assumed to agree on which 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 RECOMMENDED and 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 that 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 the protocol to abort. Examples of viable failures include when a verification check returns invalid or 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; instead, it 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 a message to be signed from an external source as input 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 will publish this commitment to every other participant. In the second round, participants will already have sufficient information to perform signing, and 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 exchanged messages are public and the channel must be reliable. However, in the setting where 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 DoS, 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. For applications that choose to apply pre-hashing, it is RECOMMENDED that they 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 each participant to check that the input message is 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. IANA Considerations

This document has no IANA actions.

# 9. References

## 9.1. Normative References


[HASH-TO-CURVE]
Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", RFC 9380, DOI 10.17487/RFC9380, August 2023, <https://www.rfc-editor.org/info/rfc9380>.

[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/info/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/info/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/info/rfc8174>.

[RISTRETTO]
de Valence, H., Grigg, J., Hamburg, M., Lovecruft, I., Tankersley, G., and F. Valsorda, "The ristretto255 and decaf448 Groups", RFC 9496, DOI 10.17487/RFC9496, December 2023, <https://www.rfc-editor.org/info/rfc9496>.

[SEC1] Standards for Efficient Cryptography, "SEC 1: Elliptic Curve Cryptography", Version 2.0, May 2009, <https://secg.org/sec1-v2.pdf>.

[SEC2] Standards for Efficient Cryptography, "SEC 2: Recommended Elliptic Curve Domain Parameters", Version 2.0, January 2010, <https://secg.org/sec2-v2.pdf>.

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


## 9.2. Informative References


[FeldmanSecretSharing]
Feldman, P., "A practical scheme for non-interactive verifiable secret sharing", IEEE, 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, "FROST: Flexible Round-Optimized Schnorr Threshold Signatures", December 2020, <https://eprint.iacr.org/2020/852.pdf>.

[MultExp] Connolly, D. and C. Gouvea, "Speeding up FROST with multi-scalar multiplication", June 2023, <https://zfnd.org/speeding-up-frost-with-multi-scalar-multiplication/>.

[Pornin22] Pornin, T., "Point-Halving and Subgroup Membership in Twisted Edwards Curves", 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/info/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/info/rfc7748>.

[ROAST] Ruffing, T., Ronge, V., Jin, E., Schneider-Bensch, J., and D. Schrder, "ROAST: Robust Asynchronous Schnorr Threshold Signatures", Paper 2022/550, DOI 10.1145/3548606, November 2022, <https://eprint.iacr.org/2022/550>.

[ShamirSecretSharing] Shamir, A., "How to share a secret", Association for Computing Machinery (ACM), Communications of the ACM, Vol. 22, Issue 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", DOI 10.1007/978-3-031-15985-5_18, August 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/info/rfc8446>.

## Appendix A. Schnorr Signature Encoding

This section describes one possible canonical encoding of FROST signatures. Using notation from Section 3 of [TLS], the encoding of a FROST signature (R, z) is as follows:

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

Where Signature.R_encoded is G.SerializeElement(R), Signature.z_encoded is G.SerializeScalar(z), and G is determined by ciphersuite.

## 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).

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 the signature R component and public key belong to the prime-order group.

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 [ShamirSecretSharing] as described in Appendices C.1 and C.2 to create secret shares of s, denoted as 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, 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 least Ns bytes of entropy.
- MAX_PARTICIPANTS, the number of shares to generate, an integer.
- MIN_PARTICIPANTS, the threshold of the secret sharing scheme, an integer.

Outputs:
- participant_private_keys, MAX_PARTICIPANTS shares of the secret key s, each a tuple consisting of the participant identifier (a NonZeroScalar) and the key share (a Scalar).
- group_public_key, public key corresponding to the group signing key, an Element.
- vss_commitment, a vector commitment of Elements in G, to each of

```
  the coefficients in the polynomial defined by secret_key_shares and
  whose first element is G.ScalarBaseMult(s).

def trusted_dealer_keygen(
        secret_key, MAX_PARTICIPANTS, MIN_PARTICIPANTS):
  # 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_key, coefficients, MAX_PARTICIPANTS)
  vss_commitment = vss_commit(coefficients):
  return participant_private_keys, vss_commitment[0], vss_commitment
```

It is assumed that 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. Furthermore, 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.

Inputs:
- s, secret value to be shared, a Scalar.
- coefficients, an array of size MIN_PARTICIPANTS - 1 with randomly generated Scalars, not including the 0th coefficient of the polynomial.
- MAX_PARTICIPANTS, the number of shares to generate, an integer less than the group order.

Outputs:
- secret_key_shares, A list of MAX_PARTICIPANTS number of secret shares, each a tuple consisting of the participant identifier (a NonZeroScalar) and the key share (a Scalar).
- coefficients, a vector of MIN_PARTICIPANTS coefficients which uniquely determine a polynomial f.

```
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.

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

Outputs:
- s, the resulting secret that was previously split into shares, a Scalar.

Errors:
- "invalid parameters", if fewer than MIN_PARTICIPANTS input shares are provided.

```
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 as 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.

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 polynomial f, each a tuple of two Scalar values representing the x and y coordinates.

```
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_coords, x)
    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 reconstruct the correct secret later.

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

```
Inputs:
- coeffs, a vector of the MIN_PARTICIPANTS coefficients that
  uniquely determine a polynomial f.

Outputs:
- vss_commitment, a vector commitment to each of the coefficients in
  coeffs, where 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 the failure should be investigated out of band.

```
Inputs:
- share_i: A tuple of the form (i, sk_i), where i indicates the
  participant identifier (a NonZeroScalar), and sk_i the
  participant's secret key, a 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
  commitment to each of the coefficients in coeffs, where each
  element of the vector commitment is an Element.

Outputs:
- True if sk_i is valid, and False otherwise.

def 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.

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

Outputs:
- PK, the public key representing the group, an Element.
- participant_public_keys, a list of MAX_PARTICIPANTS public keys PK_i for i=1,...,MAX_PARTICIPANTS, where each PK_i is the public key, an Element, for participant i.

```
def 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 described in the sections that follow.

## 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 NonZeroScalar identifier, and the following 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 the 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 NonZeroScalar 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
participant_list: 1,3
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

// Signer round one outputs
P1 hiding_nonce_randomness: 0fd2e39e111cdc266f6c0f4d0fd45c947761f1f5d
3cb583dfcb9bbaf8d4c9fec
P1 binding_nonce_randomness: 69cd85f631d5f7f2721ed5e40519b1366f340a87
c2f6856363dbdcda348a7501
```

```
P1 hiding_nonce: 812d6104142944d5a55924de6d49940956206909f2acaeedecda
2b726e630407
P1 binding_nonce: b1110165fc2334149750b28dd813a39244f315cff14d4e89e61
42f262ed83301
P1 hiding_nonce_commitment: b5aa8ab305882a6fc69cbee9327e5a45e54c08af6
1ae77cb8207be3d2ce13de3
P1 binding_nonce_commitment: 67e98ab55aa310c3120418e5050c9cf76cf387cb
20ac9e4b6fdb6f82a469f932
P1 binding_factor_input: 15d21ccd7ee42959562fc8aa63224c8851fb3ec85a3f
af66040d380fb9738673504df914fa965023fb75c25ded4bb260f417de6d32e5c442c
6ba313791cc9a4948d6273e8d3511f93348ea7a708a9b862bc73ba2a79cfdfe07729a
193751cbc973af46d8ac3440e518d4ce440a0e7d4ad5f62ca8940f32de6d8dc00fc12
c660b817d587d82f856d277ce6473cae6d2f5763f7da2e8b4d799a3f3e725d4522ec7
010000000000000000000000000000000000000000000000000000000000000000
P1 binding_factor: f2cb9d7dd9beff688da6fcc83fa89046b3479417f47f55600b
106760eb3b5603
P3 hiding_nonce_randomness: 86d64a260059e495d0fb4fcc17ea3da7452391baa
494d4b00321098ed2a0062f
P3 binding_nonce_randomness: 13e6b25afb2eba51716a9a7d44130c0dbae0004a
9ef8d7b5550c8a0e07c61775
P3 hiding_nonce: c256de65476204095ebdc01bd11dc10e57b36bc96284595b8215
222374f99c0e
P3 binding_nonce: 243d71944d929063bc51205714ae3c2218bd3451d0214dfb5ae
ec2a90c35180d
P3 hiding_nonce_commitment: cfbdb165bd8aad6eb79deb8d287bcc0ab6658ae57
fdcc98ed12c0669e90aec91
P3 binding_nonce_commitment: 7487bc41a6e712eea2f2af24681b58b1cf1da278
ea11fe4e8b78398965f13552
P3 binding_factor_input: 15d21ccd7ee42959562fc8aa63224c8851fb3ec85a3f
af66040d380fb9738673504df914fa965023fb75c25ded4bb260f417de6d32e5c442c
6ba313791cc9a4948d6273e8d3511f93348ea7a708a9b862bc73ba2a79cfdfe07729a
193751cbc973af46d8ac3440e518d4ce440a0e7d4ad5f62ca8940f32de6d8dc00fc12
c660b817d587d82f856d277ce6473cae6d2f5763f7da2e8b4d799a3f3e725d4522ec7
030000000000000000000000000000000000000000000000000000000000000000
P3 binding_factor: b087686bf35a13f3dc78e780a34b0fe8a77fef1b9938c563f5
573d71d8d7890f

// Signer round two outputs
P1 sig_share: 001719ab5a53ee1a12095cd088fd149702c0720ce5fd2f29dbecf24
b7281b603
P3 sig_share: bd86125de990acc5e1f13781d8e32c03a9bbd4c53539bbc106058bf
d14326007

sig: 36282629c383bb820a88b71cae937d41f2f2adfcc3d02e55507e2fb9e2dd3cbe
bd9d2b0844e49ae0f3fa935161e1419aab7b47d21a37ebeae1f17d4987b3160b
```

### E.2. FROST(Ed448, SHAKE256)

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

// Group input parameters
participant_list: 1,3
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

// Signer round one outputs
P1 hiding_nonce_randomness: 9cda90c98863ef3141b75f09375757286b4bc323d
d61aeb45c07de45e4937bbd
P1 binding_nonce_randomness: 781bf4881ffe1aa06f9341a747179f07a49745f8
cd37d4696f226aa065683c0a
P1 hiding_nonce: f922beb51a5ac88d1e862278d89e12c05263b945147db04b9566
acb2b5b0f7422ccea4f9286f4f80e6b646e72143eeaecc0e5988f8b2b93100
P1 binding_nonce: 1890f16a120cdeac092df29955a29c7cf29c13f6f7be60e63d6
3f3824f2d37e9c3a002dfefc232972dc08658a8c37c3ec06a0c5dc146150500
P1 hiding_nonce_commitment: 3518c2246c874569e54ab254cb1da666ca30f7879
605cc43b4d2c47a521f8b5716080ab723d3a0cd04b7e41f3cc1d3031c94ccf3829b23
fe80
P1 binding_nonce_commitment: 11b3d5220c57d02057497de3c4eebab384900206
592d877059b0a5f1d5250d002682f0e22dff096c46bb81b46d60fcfe7752ed47cea76
c3900
P1 binding_factor_input: 3832f82fda00ff5365b0376df705675b63d2a93c24c6
e81d40801ba265632be10f443f95968fadb70d10786827f30dc001c8d0f9b7c1d1b00
0e9a0f30b97fe77ef751b08d4e252a3719ae9135e7f7926f7e3b7dd6656b27089ca35
4997fe5a633aa0946c89f022462e7e9d50fd6ef313f72d956ea4571089427daa1862f
623a41625177d91e4a8f350ce9c8bd3bc7c766515dc1dd3a0eab93777526b616cccb1
48fe1e5992dc1ae705c8ba2f97ca8983328d41d375ed1e5fde5c9d672121c9e8f177f
4a1a9b2575961531b33f054451363c8f27618382cd66ce14ad93b68dac6a09f5edcbc
cc813906b3fc50b8fef1cc09757b06646f38ceed1674cd6ced28a59c93851b325c6a9
ef6a4b3b88860b7138ee246034561c7460db0b3fae501000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000
P1 binding_factor: 71966390dfdbed73cf9b79486f3b70e23b243e6c40638fb559
98642a60109daecbfcb879eed9fe7dbbed8d9e47317715a5740f772173342e00
P3 hiding_nonce_randomness: b3adf97ceea770e703ab295babf311d77e956a20d
3452b4b3344aa89a828e6df
P3 binding_nonce_randomness: 81dbe7742b0920930299197322b255734e52bbb9
1f50cfe8ce689f56fadbce31
P3 hiding_nonce: ccb5c1e82f23e0a4b966b824dbc7b0ef1cc5f56eeac2a4126e2b
2143c5f3a4d890c52d27803abcf94927faf3fc405c0b2123a57a93cefa3b00
P3 binding_nonce: e089df9bf311cf711e2a24ea27af53e07b846d09692fe11035a
1112f04d8b7462a62f34d8c01493a22b57a1cbf1f0a46c77d64d46449a90100
P3 hiding_nonce_commitment: 1254546d7d104c04e4fbcf29e05747e2edd392f67
87d05a6216f3713ef859efe573d180d291e48411e5e3006e9f90ee986ccc26b7a4249
0b80
P3 binding_nonce_commitment: 3ef0cec20be15e56b3ddcb6f7b956fca0c8f7199
0f45316b537b4f64c5e8763e6629d7262ff7cd0235d0781f23be97bf8fa8817643ea1
9cd00
P3 binding_factor_input: 3832f82fda00ff5365b0376df705675b63d2a93c24c6
e81d40801ba265632be10f443f95968fadb70d10786827f30dc001c8d0f9b7c1d1b00
0e9a0f30b97fe77ef751b08d4e252a3719ae9135e7f7926f7e3b7dd6656b27089ca35
4997fe5a633aa0946c89f022462e7e9d50fd6ef313f72d956ea4571089427daa1862f
623a41625177d91e4a8f350ce9c8bd3bc7c766515dc1dd3a0eab93777526b616cccb1
48fe1e5992dc1ae705c8ba2f97ca8983328d41d375ed1e5fde5c9d672121c9e8f177f
4a1a9b2575961531b33f054451363c8f27618382cd66ce14ad93b68dac6a09f5edcbc
cc813906b3fc50b8fef1cc09757b06646f38ceed1674cd6ced28a59c93851b325c6a9
ef6a4b3b88860b7138ee246034561c7460db0b3fae503000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000
P3 binding_factor: 236a6f7239ac2019334bad21323ec93bef2fead37bd5511435
6419f3fc1fb59f797f44079f28b1a64f51dd0a113f90f2c3a1c27d2faa4f1300

// Signer round two outputs
P1 sig_share: e1eb9bfbef792776b7103891032788406c070c5c315e3bf5d64acd4
6ea8855e85b53146150a09149665cbfec71626810b575e6f4dbe9ba3700
P3 sig_share: 815434eb0b9f9242d54b8baf2141fe28976cabe5f441ccfcd5ee7cd

```
b4b52185b02b99e6de28e2ab086c7764068c5a01b5300986b9f084f3e00

sig: cd642cba59c449dad8e896a78a60e8edfcbd9040df524370891ff8077d47ce72
1d683874483795f0d85efcbd642c4510614328605a19c6ed806ffb773b6956419537c
dfdb2b2a51948733de192dcc4b82dc31580a536db6d435e0cb3ce322fbcf9ec23362d
da27092c08767e607bf2093600
```

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

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

// Group input parameters
participant_list: 1,3
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

// Signer round one outputs
P1 hiding_nonce_randomness: f595a133b4d95c6e1f79887220c8b275ce6277e7f
68a6640e1e7140f9be2fb5c
P1 binding_nonce_randomness: 34dd1001360e3513cb37bebfabe7be4a32c5bb91
ba19fbd4360d039111f0fbdc
P1 hiding_nonce: 214f2cabb86ed71427ea7ad4283b0fae26b6746c801ce824b83c
eb2b99278c03
P1 binding_nonce: c9b8f5e16770d15603f744f8694c44e335e8faef00dad182b8d
7a34a62552f0c
P1 hiding_nonce_commitment: 965def4d0958398391fc06d8c2d72932608b1e625
5226de4fb8d972dac15fd57
P1 binding_nonce_commitment: ec5170920660820007ae9e1d363936659ef622f9
9879898db86e5bf1d5bf2a14
P1 binding_factor_input: e2a62f39eede11269e3bd5a7d97554f5ca384f9f6d3d
d9c3c0d05083c7254f572889dde2854e26377a16caf77dfee5f6be8fe5b4c80318da8
4698a4161021b033911db5ef8205362701bc9ecd983027814abee94f46d094943a2f4
b79a6e4d4603e52c435d8344554942a0a472d8ad84320585b8da3ae5b9ce31cd1903f
795c1af66de22af1a45f652cd05ee446b1b4091aaccc91e2471cd18a85a659cecd11f
0100000000000000000000000000000000000000000000000000000000000000000
P1 binding_factor: 8967fd70fa06a58e5912603317fa94c77626395a695a0e4e4e
fc4476662eba0c
P3 hiding_nonce_randomness: daa0cf42a32617786d390e0c7edfbf2efbd428037
069357b5173ae61d6dd5d5e
P3 binding_nonce_randomness: b4387e72b2e4108ce4168931cc2c7fcce5f345a5
297368952c18b5fc8473f050
P3 hiding_nonce: 3f7927872b0f9051dd98dd73eb2b91494173bbe0feb65a3e7e58
d3e2318fa40f
P3 binding_nonce: ffd79445fb8030f0a3ddd3861aa4b42b618759282bfe24f1f93
04c7009728305
P3 hiding_nonce_commitment: 480e06e3de182bf83489c45d7441879932fd7b434
a26af41455756264fbd5d6e
P3 binding_nonce_commitment: 3064746dfd3c1862ef58fc68c706da287dd92506
6865ceacc816b3a28c7b363b
P3 binding_factor_input: e2a62f39eede11269e3bd5a7d97554f5ca384f9f6d3d
```

```
d9c3c0d05083c7254f572889dde2854e26377a16caf77dfee5f6be8fe5b4c80318da8
4698a4161021b033911db5ef8205362701bc9ecd983027814abee94f46d094943a2f4
b79a6e4d4603e52c435d8344554942a0a472d8ad84320585b8da3ae5b9ce31cd1903f
795c1af66de22af1a45f652cd05ee446b1b4091aaccc91e2471cd18a85a659cecd11f
0300000000000000000000000000000000000000000000000000000000000000000
P3 binding_factor: f2c1bb7c33a10511158c2f1766a4a5fadf9f86f2a92692ed33
3128277cc31006

// Signer round two outputs
P1 sig_share: 9285f875923ce7e0c491a592e9ea1865ec1b823ead4854b48c8a462
87749ee09
P3 sig_share: 7cb211fe0e3d59d25db6e36b3fb32344794139602a7b24f1ae0dc4e
26ad7b908

sig: fc45655fbc66bbffad654ea4ce5fdae253a49a64ace25d9adb62010dd9fb2555
2164141787162e5b4cab915b4aa45d94655dbb9ed7c378a53b980a0be220a802
```

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

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

// Group input parameters
participant_list: 1,3
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

// Signer round one outputs
P1 hiding_nonce_randomness: ec4c891c85fee802a9d757a67d1252e7f4e5efb8a
538991ac18fbd0e06fb6fd3
P1 binding_nonce_randomness: 9334e29d09061223f69a09421715a347e4e6deba
77444c8f42b0c833f80f4ef9
P1 hiding_nonce: 9f0542a5ba879a58f255c09f06da7102ef6a2dec6279700c656d
58394d8facd4
P1 binding_nonce: 6513dfe7429aa2fc972c69bb495b27118c45bbc6e654bb9dc9b
e55385b55c0d7
P1 hiding_nonce_commitment: 0213b3e6298bf8ad46fd5e9389519a8665d63d98f
4ec6a1fcca434e809d2d8070e
P1 binding_nonce_commitment: 02188ff1390bf69374d7b272e454b1878ef10a6b
6ea3ff36f114b300b4dbd5233b
P1 binding_factor_input: 023a309ad94e9fe8a7ba45dfc58f38bf091959d3c99c
fbd02b4dc00585ec45ab70825371853e974bc30ac5b947b216d70461919666584c70c
51f9f56f117736c5d178dd0b521ad9c1abe98048419cbdec81504c85e12eb40e3bcb6
ec73d3fc4afd000000000000000000000000000000000000000000000000000000000
0000001
P1 binding_factor: 7925f0d4693f204e6e59233e92227c7124664a99739d2c06b8
1cf64ddf90559e
P3 hiding_nonce_randomness: c0451c5a0a5480d6c1f860e5db7d655233dca2669
fd90ff048454b8ce983367b
P3 binding_nonce_randomness: 2ba5f7793ae700e40e78937a82f407dd35e847e3
3d1e607b5c7eb6ed2a8ed799
```

```
P3 hiding_nonce: f73444a8972bcda9e506bbca3d2b1c083c10facdf4bb5d47fef7
c2dc1d9f2a0d
P3 binding_nonce: 44c6a29075d6e7e4f8b97796205f9e22062e7835141470afe94
17fd317c1c303
P3 hiding_nonce_commitment: 033ac9a5fe4a8b57316ba1c34e8a6de453033b750
e8984924a984eb67a11e73a3f
P3 binding_nonce_commitment: 03a7a2480ee16199262e648aea3acab628a53e9b
8c1945078f2ddfbdc98b7df369
P3 binding_factor_input: 023a309ad94e9fe8a7ba45dfc58f38bf091959d3c99c
fbd02b4dc00585ec45ab70825371853e974bc30ac5b947b216d70461919666584c70c
51f9f56f117736c5d178dd0b521ad9c1abe98048419cbdec81504c85e12eb40e3bcb6
ec73d3fc4afd000000000000000000000000000000000000000000000000000000000
0000003
P3 binding_factor: e10d24a8a403723bcb6f9bb4c537f316593683b472f7a89f16
6630dde11822c4

// Signer round two outputs
P1 sig_share: 400308eaed7a2ddee02a265abe6a1cfe04d946ee8720768899619cf
abe7a3aeb
P3 sig_share: 561da3c179edbb0502d941bb3e3ace3c37d122aaa46fb54499f15f3
a3331de44

sig: 026d8d434874f87bdb7bc0dfd239b2c00639044f9dcb195e9a04426f70bfa4b7
0d9620acac6767e8e3e3036815fca4eb3a3caa69992b902bcd3352fc34f1ac192f
```

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

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

// Group input parameters
participant_list: 1,3
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

// Signer round one outputs
P1 hiding_nonce_randomness: 7ea5ed09af19f6ff21040c07ec2d2adbd35b759da
5a401d4c99dd26b82391cb2
P1 binding_nonce_randomness: 47acab018f116020c10cb9b9abdc7ac10aae1b48
ca6e36dc15acb6ec9be5cdc5
P1 hiding_nonce: 841d3a6450d7580b4da83c8e618414d0f024391f2aeb511d7579
224420aa81f0
P1 binding_nonce: 8d2624f532af631377f33cf44b5ac5f849067cae2eacb88680a
31e77c79b5a80
P1 hiding_nonce_commitment: 03c699af97d26bb4d3f05232ec5e1938c12f1e6ae
97643c8f8f11c9820303f1904
P1 binding_nonce_commitment: 02fa2aaccd51b948c9dc1a325d77226e98a5a3fe
65fe9ba213761a60123040a45e
P1 binding_factor_input: 02f37c34b66ced1fb51c34a90bdae006901f10625cc0
6c4f64663b0eae87d87b4fff9b5210ffbb3c07a73a7c8935be4a8c62cf015f6cf7ade
6efac09a6513540fc3f5a816aaebc2114a811a415d7a55db7c5cbc1cf27183e79dd9d
```

```
ef941b5d4801000000000000000000000000000000000000000000000000000000000000
0000001
P1 binding_factor: 3e08fe561e075c653cbfd46908a10e7637c70c74f0a77d5fd4
5d1a750c739ec6
P3 hiding_nonce_randomness: e6cc56ccbd0502b3f6f831d91e2ebd01c4de0479e
0191b66895a4ffd9b68d544
P3 binding_nonce_randomness: 7203d55eb82a5ca0d7d83674541ab55f6e76f1b8
5391d2c13706a89a064fd5b9
P3 hiding_nonce: 2b19b13f193f4ce83a399362a90cdc1e0ddcd83e57089a7af0bd
ca71d47869b2
P3 binding_nonce: 7a443bde83dc63ef52dda354005225ba0e553243402a4705ce2
8ffaafe0f5b98
P3 hiding_nonce_commitment: 03077507ba327fc074d2793955ef3410ee3f03b82
b4cdc2370f71d865beb926ef6
P3 binding_nonce_commitment: 02ad53031ddfbbacfc5fbda3d3b0c2445c8e3e99
cbc4ca2db2aa283fa68525b135
P3 binding_factor_input: 02f37c34b66ced1fb51c34a90bdae006901f10625cc0
6c4f64663b0eae87d87b4fff9b5210ffbb3c07a73a7c8935be4a8c62cf015f6cf7ade
6efac09a6513540fc3f5a816aaebc2114a811a415d7a55db7c5cbc1cf27183e79dd9d
ef941b5d4801000000000000000000000000000000000000000000000000000000000000
0000003
P3 binding_factor: 93f79041bb3fd266105be251adaeb5fd7f8b104fb554a4ba9a
0becea48ddbfd7

// Signer round two outputs
P1 sig_share: c4fce1775a1e141fb579944166eab0d65eefe7b98d480a569bbbfcb
14f91c197
P3 sig_share: 0160fd0d388932f4826d2ebcd6b9eaba734f7c71cf25b4279a4ca25
81e47b18d

sig: 0205b6d04d3774c8929413e3c76024d54149c372d57aae62574ed74319b5ea14
d0c65dde8492a7471437e6c2fe3da49b90d23f642b5c6dbe7e36089f096dd97324
```

## 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 input and contributions.

## 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