Workgroup: CFRG
Internet-Draft: draft-irtf-cfrg-frost-05
Published: 31 May 2022
Intended Status: Informational
Expires: 2 December 2022
A   D. Connolly
        uZcash Foundation
        t
        h
        o
        r
        s
        :
         C. Komlo
         University of Waterloo, Zcash Foundation
         I. Goldberg   C. A. Wood
         University of Waterloo   Cloudflare

Two-Round Threshold Schnorr Signatures with FROST

Abstract

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

Discussion Venues

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

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

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

Status of This Memo

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

This Internet-Draft will expire on 2 December 2022.

Copyright Notice

Table of Contents

1. Introduction

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

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

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

This document presents a variant of a Flexible Round-Optimized Schnorr Threshold (FROST) signature scheme originally defined in [FROST20]. FROST reduces network overhead during threshold signing operations while employing a novel technique to protect against forgery attacks applicable to prior Schnorr-based threshold signature constructions. The variant of FROST presented in this document requires two rounds to compute a signature, and implements signing efficiency improvements described by [Schnorr21]. Single-round signing with FROST is out of scope.

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

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 B.

1.1. Change Log

draft-05

- *Update test vectors to include version string (#202, #203)
- *Rename THRESHOLD_LIMIT to MIN_SIGNERS (#192)
- *Use non-contiguous signers for the test vectors (#187)
- *Add more reasoning why the coordinator MUST abort (#183)
- *Add a function to generate nonces (#182)

* Add MUST that all participants have the same view of VSS commitment (#174)

* Use THRESHOLD_LIMIT instead of t and MAX_SIGNERS instead of n (#171)

* Specify what the dealer is trusted to do (#166)

* Clarify types of NUM_SIGNERS and THRESHOLD_LIMIT (#165)

* Assert that the network channel used for signing should be authenticated (#163)

* Remove wire format section (#156)

* Update group commitment derivation to have a single scalarmul (#150)

* Use RandomNonzeroScalar for single-party Schnorr example (#148)

* Fix group notation and clarify member functions (#145)

* Update existing implementations table (#136)

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

draft-04

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

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

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

* Split test vectors into separate files (#129)

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

* Add failure checks (#127)

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

* Define cofactor checks for verification (#118)

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

draft-03

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

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

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

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

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

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

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

* Apply various editorial improvements.

draft-02

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

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

* Refactor document for editorial clarity.

draft-01

* Specify operations, notation and cryptographic dependencies.

draft-00

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

2. Conventions and Definitions

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

The following notation and terminology are used throughout this document.

* A participant is an entity that is trusted to hold a secret share.

* MAX_SIGNERS denotes the number of participants, and the number of shares that s is split into. This value MUST NOT exceed 2^16-1.

* MIN_SIGNERS denotes the threshold number of participants required to issue a signature. More specifically, at least MIN_SIGNERS shares must be combined to issue a valid signature. This value MUST NOT exceed p.

- *NUM_SIGNERS denotes the number of signers that participate in an invocation of FROST signing, where MIN_SIGNERS <= NUM_SIGNERS <= MAX_SIGNERS. This value MUST NOT exceed p.

- *len(x) is the length of integer input x as an 8-byte, big-endian integer.

- *encode_uint16(x): Convert two byte unsigned integer (uint16) x to a 2-byte, big-endian byte string. For example, encode_uint16(310) = [0x01, 0x36].

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

- *|| denotes concatenation, i.e., x || y = xy.

- *nil denotes an empty byte string.

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

3. Cryptographic Dependencies

FROST signing depends on the following cryptographic constructs:

- *Prime-order Group, Section 3.1;

- *Cryptographic hash function, Section 3.2;

These are described in the following sections.

3.1. Prime-Order Group

FROST depends on an abelian group of prime order p. We represent this group as the object G that additionally defines helper functions described below. The group operation for G is addition + with identity element I. For any elements A and B of the group G, A + B = B + A is also a member of G. Also, for any A in G, there exists an element -A such that A + (-A) = (-A) + A = I. For convenience, we use - to denote the subtraction, e.g., A - B = A + (-B). Scalar multiplication is equivalent to the repeated application of the group operation on an element A with itself r-1 times, this is denoted as r*A = A + ... + A. For any element A, p * A = I. We denote B as a fixed generator of the group. Scalar base multiplication is equivalent to the repeated application of the group operation B with itself r-1 times, this is denoted as ScalarBaseMult(r). The set of scalars corresponds to GF(p), which refer to as the scalar field. This document uses types Element and Scalar to denote elements of the group G and its set of scalars, respectively. We denote equality comparison as == and assignment of values by =.

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).

- *RandomNonzeroScalar(): Outputs a random non-zero Scalar element in GF(p).

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

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

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

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

3.2. Cryptographic Hash Function

FROST requires the use of a cryptographically secure hash function, generically written as H, which functions effectively as a random oracle. For concrete recommendations on hash functions which SHOULD be used in practice, see Section 6. Using H, we introduce three separate domain-separated hashes, H1, H2, H3, and H4, where H1, H2, and H4 map arbitrary byte strings to Scalar elements of the prime-order group scalar field, and H3 is an alias for H with a domain separator. The details of H1, H2, H3, and H4 vary based on ciphersuite. See Section 6 for more details about each.

4. Helper functions

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

- *Nonce generation, Section 4.1

- *Schnorr signatures, Section 4.2;

- *Polynomial operations, Section 4.3;

- *Encoding operations, Section 4.4;

- *Signature binding Section 4.5, group commitment Section 4.6, and challenge computation Section 4.7

These sections describes these operations in more detail.

4.1. Nonce generation

To hedge against a bad RNG, we generate nonces by sourcing fresh randomness and combine with the secret key, to create a domain-separated hash function from the ciphersuite hash function H, H4:

```
nonce_generate(secret):

Inputs:
- secret, a Scalar

Outputs: nonce, a Scalar

def nonce_generate(secret):
  k_enc = random_bytes(32)
  secret_enc = G.SerializeScalar(secret)
  return H4(k_enc || secret_enc)
```

4.2. Schnorr Signature Operations

In the single-party setting, a Schnorr signature is generated with the following operation.

```
schnorr_signature_generate(msg, sk):

Inputs:
- msg, message to be signed, a byte string
- sk, private key, a Scalar

Outputs: signature (R, z), a pair consisting of (Element, Scalar) valu

def schnorr_signature_generate(msg, sk):
  PK = G.ScalarBaseMult(sk)
  k = nonce_generate(sk)
  R = G.ScalarBaseMult(k)

  comm_enc = G.SerializeElement(R)
  pk_enc = G.SerializeElement(PK)
  challenge_input = comm_enc || pk_enc || msg
  c = H2(challenge_input)

  z = k + (c * sk)
  return (R, z)
```

The corresponding verification operation is as follows. By definition, this function assumes that all group Elements passed as input, including the signature R component and public key, belong to prime-order group G. Some ciphersuites defined in Section 6 operate in settings where some inputs may not belong to this prime-order group, e.g., because they operate over an elliptic curve with a cofactor larger than 1. In these settings, input validation is required before invoking this verification function.

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

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

Outputs: 1 if signature is valid, and 0 otherwise

def schnorr_signature_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 + (c * PK)
  check = l - r
  return check == G.Identity()
```

4.3. Polynomial Operations

This section describes operations on and associated with polynomials over Scalars that are used in the main signing protocol. A polynomial of degree t is represented as a list of t coefficients, where the constant term of the polynomial is in the first position and the highest-degree coefficient is in the last position. A point on the polynomial is a tuple (x, y), where y = f(x). For notational convenience, we refer to the x-coordinate and y-coordinate of a point p as p.x and p.y, respectively.

4.3.1. Evaluation of a polynomial

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

```
polynomial_evaluate(x, coeffs):

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

Outputs: Scalar result of the polynomial evaluated at input x

def polynomial_evaluate(x, coeffs):
  value = 0
  for (counter, coeff) in coeffs.reverse():
    if counter == coeffs.len() - 1:
      value += coeff // add the constant term
    else:
      value += coeff
      value *= x
  return value
```

4.3.2. Lagrange coefficients

The function derive_lagrange_coefficient derives a Lagrange coefficient to later perform polynomial interpolation, and is

provided a set of x-coordinates as input. Lagrange coefficients are used in FROST to evaluate a polynomial f at point 0, i.e., f(0), given a set of t other points.

```
derive_lagrange_coefficient(x_i, L):

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

Outputs: L_i, the i-th Lagrange coefficient

Errors:
- "invalid parameters", if any coordinate is equal to 0

def derive_lagrange_coefficient(x_i, L):
  if x_i = 0:
    raise "invalid parameters"
  for x_j in L:
    if x_j = 0:
      raise "invalid parameters"

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

  L_i = numerator / denominator
  return L_i
```

4.3.3. Deriving the constant term of a polynomial

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

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

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

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

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

  return f_zero
```

Note that these coefficients can be computed once and then stored, as these values remain constant across FROST signing sessions.

4.4. Commitment List Encoding

This section describes the subroutine used for encoding a list of signer commitments into a bytestring that is used in the FROST protocol.

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

Outputs: A byte string containing the serialized representation of com

def encode_group_commitment_list(commitment_list):
  encoded_group_commitment = nil
  for (identifier, hiding_nonce_commitment, binding_nonce_commitment)
    encoded_commitment = encode_uint16(identifier) ||
                         G.SerializeElement(hiding_nonce_commitment) |
                         G.SerializeElement(binding_nonce_commitment)
    encoded_group_commitment = encoded_group_commitment || encoded_com
  return encoded_group_commitment
```

4.5. Binding Factor Computation

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

```
Inputs:
- encoded_commitment_list, an encoded commitment list (as computed
  by encode_group_commitment_list)
- msg, the message to be signed.

Outputs: A Scalar representing the binding factor

def compute_binding_factor(encoded_commitment_list, msg):
  msg_hash = H3(msg)
  rho_input = encoded_commitment_list || msg_hash
  binding_factor = H1(rho_input)
  return binding_factor
```

4.6. Group Commitment Computation

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

```
Inputs:
- commitment_list =
   [(i, hiding_nonce_commitment_i, binding_nonce_commitment_i), ...],
  of commitments issued by each signer, where each element in the list
  indicates the signer identifier i and their two commitment Element v
  (hiding_nonce_commitment_i, binding_nonce_commitment_i). This list M
  sorted in ascending order by signer identifier.
- binding_factor, a Scalar

Outputs: An Element in G representing the group commitment

def compute_group_commitment(commitment_list, binding_factor):
  group_hiding_commitment = G.Identity()
  group_binding_commitment = G.Identity()

  for (_, hiding_nonce_commitment, binding_nonce_commitment) in commit
    group_hiding_commitment = group_hiding_commitment + hiding_nonce_c
    group_binding_commitment = group_binding_commitment + binding_nonc
  return (group_hiding_commitment + group_binding_commitment * binding
```

4.7. Signature Challenge Computation

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

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

Outputs: A Scalar representing the challenge

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

5. Two-Round FROST Signing Protocol

We now present the two-round variant of the FROST threshold signature protocol for producing Schnorr signatures. It involves signer participants and a Coordinator. Signing participants are entities with signing key shares that participate in the threshold signing protocol. The Coordinator is an entity with the following responsibilities:

1. Determining which signers will participate (at least MIN_SIGNERS in number);

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

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

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

Because key generation is not specified, all signers are assumed to have the (public) group state that we refer to as "group info" below, and their corresponding signing key shares.

In particular, it is assumed that the Coordinator and each signing participant P_i knows the following group info:

- Group public key, an Element in G, denoted PK = G.ScalarMultBase(s), corresponding to the group secret key s, which is a Scalar. PK is an output from the group's key generation protocol, such as trusted_dealer_keygenor a DKG.

- Public keys for each signer, denoted PK_i = G.ScalarMultBase(), which are similarly outputs from the group's key generation protocol, Elements in G.

And that each participant with identifier i where i is an integer in the range between 1 and MAX_SIGNERS additionally knows the following:

- Participant is signing key share sk_i, which is the i-th secret share of s, a Scalar.

The exact key generation mechanism is out of scope for this specification. In general, key generation is a protocol that outputs (1) a shared, group public key PK owned by each Signer, and (2) individual shares of the signing key owned by each Signer. In general, two possible key generation mechanisms are possible, one that requires a single, trusted dealer, and the other which requires performing a distributed key generation protocol. We highlight key generation mechanism by a trusted dealer in Appendix B, for reference.

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

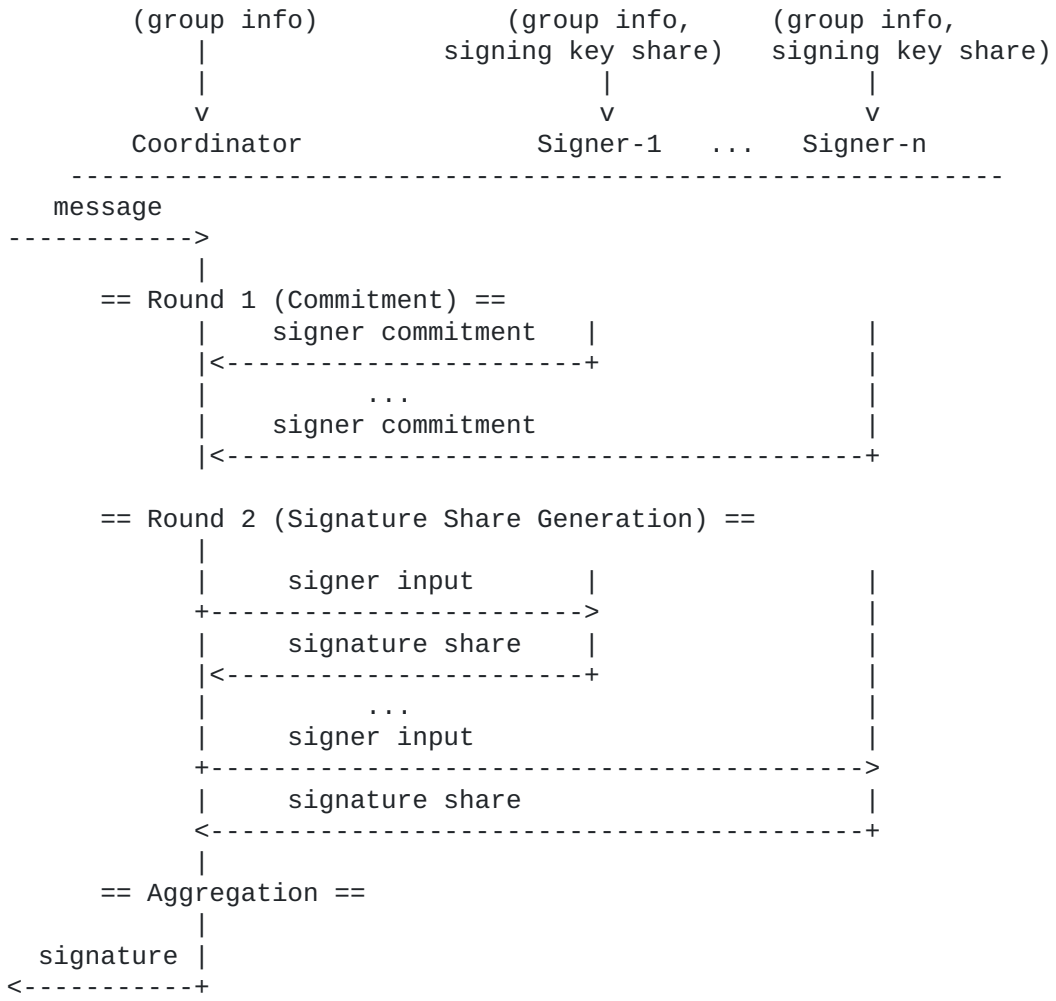

Figure 1: FROST signature overview

Details for round one are described in Section 5.1, and details for round two are described in Section 5.2. The final Aggregation step is described in Section 5.3.

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

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

5.1. Round One - Commitment

Round one involves each signer 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 signer in round one generates a nonce nonce = (hiding_nonce, binding_nonce) and commitment comm = (hiding_nonce_commitment, binding_nonce_commitment).

Inputs: sk_i, the secret key share, a Scalar

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

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

The private output nonce from Participant P_i is stored locally and kept private for use in the second round. This nonce MUST NOT be reused in more than one invocation of FROST, and it MUST be generated from a source of secure randomness. The public output comm from Participant P_i is sent to the Coordinator.

5.2. Round Two - Signature Share Generation

In round two, the Coordinator is responsible for sending the message to be signed, and for choosing which signers will participate (of number at least MIN_SIGNERS). 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 signer the message to be signed along with the set of signing commitments for all other signers in the participant list. Each signer 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. Applications which require that signers not process arbitrary input messages are also required to also perform relevant application-layer input validation checks; see Section 7.4 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 signer. Note identifier will never e
- sk_i, Signer secret key share, a Scalar.
- group_public_key, public key corresponding to the group signing key,
  an Element in G.
- nonce_i, pair of Scalar values (hiding_nonce, binding_nonce) generat
  round one.
- msg, the message to be signed (sent by the Coordinator).
- commitment_list =
    [(j, hiding_nonce_commitment_j, binding_nonce_commitment_j), ...],
  list of commitments issued in Round 1 by each signer and sent by the
  Each element in the list indicates the signer identifier j and their
  Element values (hiding_nonce_commitment_j, binding_nonce_commitment_
  This list MUST be sorted in ascending order by signer identifier.

Outputs: a Scalar value representing the signature share

def sign(identifier, sk_i, group_public_key, nonce_i, msg, commitment_
  # Encode the commitment list
  encoded_commitments = encode_group_commitment_list(commitment_list)

  # Compute the binding factor
  binding_factor = compute_binding_factor(encoded_commitments, msg)

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

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

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

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

  return sig_share
```

The output of this procedure is a signature share. Each signer then sends these shares back to the Coordinator. Each signer MUST delete the nonce and corresponding commitment after this round completes.

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

5.3. Signature Share Verification and Aggregation

After signers perform round two and send their signature shares to the Coordinator, the Coordinator verifies each signature share for correctness. In particular, for each signer, the Coordinator uses commitment pairs generated during round one and the signature share generated during round two, along with other group parameters, to

check that the signature share is valid using the following procedure.

```
Inputs:
- identifier, Identifier i of the signer. Note identifier will never e
- PK_i, the public key for the ith signer, where PK_i = G.ScalarBaseMu
  an Element in G
- comm_i, pair of Element values in G (hiding_nonce_commitment, bindin
  generated in round one from the ith signer.
- sig_share_i, a Scalar value indicating the signature share as produc
  round two from the ith signer.
- commitment_list =
    [(j, hiding_nonce_commitment_j, binding_nonce_commitment_j), ...],
  list of commitments issued in Round 1 by each signer, where each ele
  in the list indicates the signer identifier j and their two commitme
  Element values (hiding_nonce_commitment_j, binding_nonce_commitment_
  This list MUST be sorted in ascending order by signer identifier.
- group_public_key, public key corresponding to the group signing key,
  an Element in G.
- msg, the message to be signed.

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

def verify_signature_share(identifier, PK_i, comm_i, sig_share_i, comm
                           group_public_key, msg):
  # Encode the commitment list
  encoded_commitments = encode_group_commitment_list(commitment_list)

  # Compute the binding factor
  binding_factor = compute_binding_factor(encoded_commitments, msg)

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

  # Compute the commitment share
  (hiding_nonce_commitment, binding_nonce_commitment) = comm_i
  comm_share = hiding_nonce_commitment + (binding_nonce_commitment * b

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

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

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

  return l == r
```

If any signature share fails to verify, i.e., if verify_signature_share returns False for any signer share, the Coordinator MUST abort the protocol for correctness reasons. Excluding one signer means that their nonce will not be included in the joint response z and consequently the output signature will not verify.

Otherwise, if all signer shares are valid, the Coordinator performs the aggregate operation and publishes the resulting signature.

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

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

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

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

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

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

6. Ciphersuites

A FROST ciphersuite must specify the underlying prime-order group details and cryptographic hash function. Each ciphersuite is denoted as (Group, Hash), e.g., (ristretto255, SHA-512). This section contains some ciphersuites.

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

The DeserializeElement and DeserializeScalar functions instantiated for a particular prime-order group corresponding to a ciphersuite MUST adhere to the description in Section 3.1. Validation steps for these functions are described for each the ciphersuites below. Future ciphersuites MUST describe how input validation is done for DeserializeElement and DeserializeScalar.

6.1. FROST(Ed25519, SHA-512)

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

- *Group: edwards25519 [RFC8032]
  - -Order: 2^252 + 27742317777372353535851937790883648493 (see [RFC7748])

- Identity: As defined in [RFC7748].

- RandomScalar: Implemented by generating a random 32-byte string and invoking DeserializeScalar on the result.

- RandomNonZeroScalar: Implemented by generating a random 32-byte string that is not equal to the all-zero string and invoking DeserializeScalar on the result.

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

- DeserializeElement: Implemented as specified in [RFC8032], Section 5.1.3. Additionally, this function validates that the resulting element is not the group identity element.

- SerializeScalar: Implemented by outputting the little-endian 32-byte encoding of the Scalar value.

- DeserializeScalar: Implemented by attempting to deserialize a Scalar from a 32-byte string. This function can fail if the input does not represent a Scalar between the value 0 and G.Order() - 1.

* Hash (H): SHA-512, and Nh = 64.

  - H1(m): Implemented by computing H("rho" || m), interpreting the lower 32 bytes as a little-endian integer, and reducing the resulting integer modulo L = 2^252+27742317777372353535851937790883648493.

  - H2(m): Implemented by computing H(m), interpreting the lower 32 bytes as a little-endian integer, and reducing the resulting integer modulo L = 2^252+27742317777372353535851937790883648493.

  - H3(m): Implemented as an alias for H, i.e., H(m).

  - H4(m): Implemented by computing H("nonce" || m), interpreting the lower 32 bytes as a little-endian integer, and reducing the resulting integer modulo L = 2^252+27742317777372353535851937790883648493.

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

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-v5".

* Group: ristretto255 [RISTRETTO]

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

  - Identity: As defined in [RISTRETTO].

- RandomScalar: Implemented by generating a random 32-byte string and invoking DeserializeScalar on the result.

- RandomNonZeroScalar: Implemented by generating a random 32-byte string that is not equal to the all-zero string and invoking DeserializeScalar on the result.

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

- DeserializeElement: Implemented using the 'Decode' function from [RISTRETTO].

- SerializeScalar: Implemented by outputting the little-endian 32-byte encoding of the Scalar value.

- DeserializeScalar: Implemented by attempting to deserialize a Scalar from a 32-byte string. This function can fail if the input does not represent a Scalar between the value 0 and G.Order() - 1.

* Hash (H): SHA-512, and Nh = 64.

- 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 || "digest" || m).

- H4(m): Implemented by computing H(contextString || "nonce" || m) and mapping the output to a Scalar as described in [RISTRETTO], Section 4.4.

6.3. FROST(Ed448, SHAKE256)

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

* Group: edwards448 [RFC8032]

- Order: 2^446 - 138180668098951153520073867485154268803366924748821786098945475038885

- Identity: As defined in [RFC7748].

- RandomScalar: Implemented by generating a random 48-byte string and invoking DeserializeScalar on the result.

- RandomNonZeroScalar: Implemented by generating a random 48-byte string that is not equal to the all-zero string and invoking DeserializeScalar on the result.

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

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

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

- DeserializeScalar: Implemented by attempting to deserialize a Scalar from a 48-byte string. This function can fail if the input does not represent a Scalar between the value 0 and G.Order() - 1.

* Hash (H): SHAKE256, and Nh = 117.

  - H1(m): Implemented by computing H("rho" || m), interpreting the lower 57 bytes as a little-endian integer, and reducing the resulting integer modulo L = 2^446 - 138180668098951153520073867485154268803366924748821786098945 47503885.

  - H2(m): Implemented by computing H(m), interpreting the lower 57 bytes as a little-endian integer, and reducing the resulting integer modulo L = 2^446 - 138180668098951153520073867485154268803366924748821786098945 47503885.

  - H3(m): Implemented as an alias for H, i.e., H(m).

  - H4(m): Implemented by computing H("nonce" || m), interpreting the lower 57 bytes as a little-endian integer, and reducing the resulting integer modulo L = 2^446 - 138180668098951153520073867485154268803366924748821786098945 47503885.

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

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-v5".

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

  - Order: 0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc63 2551

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

- RandomScalar: Implemented by generating a random 32-byte string and invoking DeserializeScalar on the result.

- RandomNonZeroScalar: Implemented by generating a random 32-byte string that is not equal to the all-zero string and invoking DeserializeScalar on the result.

- SerializeElement: Implemented using the compressed Elliptic-Curve-Point-to-Octet-String method according to [SECG].

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

- SerializeScalar: Implemented using the Field-Element-to-Octet-String conversion according to [SECG].

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

* Hash (H): SHA-256, and Nh = 32.

  - H1(m): Implemented using hash_to_field from [HASH-TO-CURVE], Section 5.3 using L = 48, expand_message_xmd with SHA-256, DST = contextString || "rho", and prime modulus equal to Order().

  - H2(m): Implemented using hash_to_field from [HASH-TO-CURVE], Section 5.3 using L = 48, expand_message_xmd with SHA-256, DST = contextString || "chal", and prime modulus equal to Order().

  - H3(m): Implemented by computing H(contextString || "digest" || m).

  - H4(m): Implemented using hash_to_field from [HASH-TO-CURVE], Section 5.3 using L = 48, expand_message_xmd with SHA-256, DST = contextString || "nonce", and prime modulus equal to Order().

7. Security Considerations

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

* Trusted dealer. The dealer that performs key generation is trusted to follow the protocol, although participants still are able to verify the consistency of their shares via a VSS (verifiable secret sharing) step; see Appendix B.2.

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

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

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

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

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

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

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

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

7.1. Nonce Reuse Attacks

Nonces generated by each participant in the first round of signing must be sampled uniformly at random and cannot be derived from some deterministic function. This is to avoid replay attacks initiated by other signers, which allows for a complete key-recovery attack. The Coordinator MAY further hedge against nonce reuse attacks by tracking signer nonce commitments used for a given group key, at the cost of additional state.

7.2. Protocol Failures

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

7.3. Removing the Coordinator Role

In some settings, it may be desirable to omit the role of the Coordinator entirely. Doing so does not change the security implications of FROST, but instead simply requires each participant to communicate with all other participants. We loosely describe how

to perform FROST signing among signers without this coordinator role. We assume that every participant receives as input from an external source the message to be signed prior to performing the protocol.

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

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

7.4. Input Message Validation

Some applications may require that signers only process messages of a certain structure. For example, in digital currency applications wherein multiple signers may collectively sign a transaction, it is reasonable to require that each signer check the input message to be a syntactically valid transaction. As another example, use of threshold signatures in TLS [TLS] to produce signatures of transcript hashes might require that signers check that the input message is a valid TLS transcript from which the corresponding transcript hash can be derived.

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 signers do not operate as signing oracles for arbitrary messages.

8. Contributors

*Isis Lovecruft

*T. Wilson-Brown

*Alden Torres

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", Work in Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-14, 18 February 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-14>.

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

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

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://doi.org/10.17487/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://doi.org/10.17487/RFC8174>.

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

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

[x9.62] ANSI, "Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62-1998, September 1998.

9.2. Informative References

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

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

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

4086, DOI 10.17487/RFC4086, June 2005, <https://doi.org/10.17487/RFC4086>.

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

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

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

Appendix A. Acknowledgments

This document was improved based on input and contributions by the Zcash Foundation engineering team.

Appendix B. Trusted Dealer Key Generation

One possible key generation mechanism is to depend on a trusted dealer, wherein the dealer generates a group secret s uniformly at random and uses Shamir and Verifiable Secret Sharing as described in Appendix B.1 and Appendix B.2 to create secret shares of s to be sent to all other participants. We highlight at a high level how this operation can be performed.

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

```
Inputs:
- s, a group secret, Scalar, that MUST be derived from at least Ns byt
- MAX_SIGNERS, the number of shares to generate, an integer
- MIN_SIGNERS, the threshold of the secret sharing scheme, an integer

Outputs:
- signer_private_keys, MAX_SIGNERS shares of the secret key s, each a
- vss_commitment, a vector commitment of Elements in G, to each of the
  in the polynomial defined by secret_key_shares and whose constant te
  G.ScalarBaseMult(s).

def trusted_dealer_keygen(s, MAX_SIGNERS, MIN_SIGNERS):
  signer_private_keys, coefficients = secret_share_shard(secret_key, M
  vss_commitment = vss_commit(coefficients):
  PK = G.ScalarBaseMult(secret_key)
  return signer_private_keys, vss_commitment
```

It is assumed the dealer then sends one secret key share to each of the NUM_SIGNERS participants, along with C. After receiving their secret key share and C, participants MUST abort if they do not have the same view of C. Otheriwise, each participant MUST perform vss_verify(secret_key_share_i, C), 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.

B.1. Shamir Secret Sharing

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

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

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

secret_share_shard(s, MAX_SIGNERS, MIN_SIGNERS):

Inputs:
- s, secret value to be shared, a Scalar
- MAX_SIGNERS, the number of shares to generate, an integer
- MIN_SIGNERS, the threshold of the secret sharing scheme, an integer

Outputs:
- secret_key_shares, A list of MAX_SIGNERS number of secret shares, wh
  consisting of the participant identifier and the key share, each of
- coefficients, a vector of the t coefficients which uniquely determin
  a polynomial f.

Errors:
- "invalid parameters", if MIN_SIGNERS > MAX_SIGNERS or if MIN_SIGNERS

```
def secret_share_shard(s, MAX_SIGNERS, MIN_SIGNERS):
  if MIN_SIGNERS > MAX_SIGNERS:
    raise "invalid parameters"
  if MIN_SIGNERS < 2:
    raise "invalid parameters"

  # Generate random coefficients for the polynomial, yielding
  # a polynomial of degree (MIN_SIGNERS - 1)
  coefficients = [s]
  for i in range(1, MIN_SIGNERS):
    coefficients.append(G.RandomScalar())

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

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

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

The procedure for combining a shares list of length MIN_SIGNERS to recover the secret s is as follows.

```
secret_share_combine(shares):

Inputs:
- shares, a list of at minimum MIN_SIGNERS secret shares, each a tuple

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

Errors:
- "invalid parameters", if less than MIN_SIGNERS input shares are prov

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

B.2. Verifiable Secret Sharing

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

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

```
vss_commit(coeffs):

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

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

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

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

```
vss_verify(share_i, vss_commitment):

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

Outputs: 1 if sk_i is valid, and 0 otherwise

vss_verify(share_i, commitment)
  (i, sk_i) = share_i
  S_i = ScalarBaseMult(sk_i)
  S_i' = G.Identity()
  for j in range(0, MIN_SIGNERS-1):
    S_i' += vss_commitment[j] * i^j
  if S_i == S_i':
    return 1
  return 0
```

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

```
derive_group_info(MAX_SIGNERS, MIN_SIGNERS, vss_commitment):

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

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

derive_group_info(MAX_SIGNERS, MIN_SIGNERS, vss_commitment)
  PK = vss_commitment[0]
  signer_public_keys = []
  for i in range(1, MAX_SIGNERS+1):
    PK_i = G.Identity()
    for j in range(0, MIN_SIGNERS):
      PK_i += vss_commitment[j] * i^j
    signer_public_keys.append(PK_i)
  return PK, signer_public_keys
```

Appendix C. 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_SIGNERS, MIN_SIGNERS, and NUM_SIGNERS.

- Group input parameters: This lists the group secret key and shared public key, generated by a trusted dealer as described in Appendix B, as well as the input message to be signed. All values are encoded as hexadecimal strings.

- Signer input parameters: This lists the signing key share for each of the NUM_SIGNERS signers.

- Round one parameters and outputs: This lists the NUM_SIGNERS participants engaged in the protocol, identified by their integer identifier, the hiding and binding commitment values produced in Section 5.1, as well as the resulting group binding factor input, computed in part from the group commitment list encoded as described in Section 4.4, and group binding factor as computed in Section 5.2).

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

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

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

```
// Configuration information
MAX_SIGNERS: 3
MIN_SIGNERS: 2
NUM_SIGNERS: 2

// Group input parameters
group_secret_key: 7b1c33d3f5291d85de664833beb1ad469f7fb6025a0ec78b3a7
90c6e13a98304
group_public_key: 15d21ccd7ee42959562fc8aa63224c8851fb3ec85a3faf66040
d380fb9738673
message: 74657374

// Signer input parameters
S1 signer_share: 929dcc590407aae7d388761cddb0c0db6f5627aea8e217f4a033
f2ec83d93509
S2 signer_share: a91e66e012e4364ac9aaa405fcafd370402d9859f7b6685c07ee
d76bf409e80d
S3 signer_share: d3cb090a075eb154e82fdb4b3cb507f110040905468bb9c46da8
bdea643a9a02

// Round one parameters
participants: 1,3
group_binding_factor_input: 000165600ffa0fa6af339deb0c9237e60defc4444
c90c117351916bb56d46d0e4a866c0f71de419ec78fad43cdd8075386f8dc88b6c29b
51164d0ea80f83b838b5ec00030240580b6b54457ffe21200a16dad7b23532739340b
46cbbb75257ae1ee176855fdea859ce7a61dae083a3d3de4e3ca5133c18a7d856442f
223c55934bfbc7d7ee26b0dd4af7e749aa1a8ee3c10ae9923f618980772e473f8819a
5d4940e0db27ac185f8a0e1d5f84f88bc887fd67b143732c304cc5fa9ad8e6f57f500
28a8ff
group_binding_factor: d715d01b68ed6744820dfe5630e7d51e986fb5dd0a8d0c1
6547d5382e9afeb0f

// Signer round one outputs
S1 hiding_nonce: 8c76af04340e83bb5fc427c117d38347fc8ef86d5397feea9aa6
412d96c05b0a
S1 binding_nonce: 14a37ddbeae8d9e9687369e5eb3c6d54f03dc19d76bb54fb542
5131bc37a600b
S1 hiding_nonce_commitment: 65600ffa0fa6af339deb0c9237e60defc4444c90c
117351916bb56d46d0e4a86
S1 binding_nonce_commitment: 6c0f71de419ec78fad43cdd8075386f8dc88b6c2
9b51164d0ea80f83b838b5ec
S3 hiding_nonce: 5ca39ebab6874f5e7b5089f3521819a2aa1e2cf738bae6974ee8
0555de2ef70e
S3 binding_nonce: 0afe3650c4815ff37becd3c6948066e906e929ea9b8f546c74e
10002dbcc150c
S3 hiding_nonce_commitment: 0240580b6b54457ffe21200a16dad7b2353273934
0b46cbbb75257ae1ee17685
S3 binding_nonce_commitment: 5fdea859ce7a61dae083a3d3de4e3ca5133c18a7
d856442f223c55934bfbc7d7

// Round two parameters
participants: 1,3

// Signer round two outputs
S1 sig_share: 4369474a398aa10357b60d683da91ea6a767dcf53fd541a8ed6b4d7
80827ea0a
S3 sig_share: 32fcc690d926075e45d2dfb746bab71447943cddbefe80d122c3917
4aa2e1004
```

sig: 2b8d9c6995333c5990e3a3dd6568785539d3322f7f0376452487ea35cfda587b
75650edb12b1a8619c88ed1f8463d6baeefb18d3fed3c279102fdfecb255fa0e

C.2. FROST(Ed448, SHAKE256)

```
// Configuration information
MAX_SIGNERS: 3
MIN_SIGNERS: 2
NUM_SIGNERS: 2

// Group input parameters
group_secret_key: 6298e1eef3c379392caaed061ed8a31033c9e9e3420726f23b4
04158a401cd9df24632adfe6b418dc942d8a091817dd8bd70e1c72ba52f3c00
group_public_key: 1588564c56a8edb53b55399df5b65fd2abe777717baa2ef440b
13fe13b7ce077347f5e4346ab4475f9258fb947978b0123884832a46c6be800
message: 74657374

// Signer input parameters
S1 signer_share: 4a2b2f5858a932ad3d3b18bd16e76ced3070d72fd79ae4402df2
01f525e754716a1bc1b87a502297f2a99d89ea054e0018eb55d39562fd0100
S2 signer_share: 2503d56c4f516444a45b080182b8a2ebbe4d9b2ab509f25308c8
8c0ea7ccdc44e2ef4fc4f63403a11b116372438a1e287265cadeff1fcb0700
S3 signer_share: 00db7a8146f995db0a7cf844ed89d8e94c2b5f259378ff66e39d
172828b264185ac4decf7219e4aa4478285b9c0eef4fccdf3eea69dd980d00

// Round one parameters
participants: 1,3
group_binding_factor_input: 0001473d732afb8ee9647e1ace23bf148a099c356
e964b3a22652c89008792d6a699a9db4b830b100ce16ddef6d9780d1f8b2c099dc9a4
0c827a0025a10958a534d96765609b368f35b6d36cd45db6c5fbc00aee224671b2a37
fe736dbadfb5e67161b4184d1628b378bdde07afc49a36300ec000003b80d36d789a1
bbc65e6855067b90526cbe81d0bf3da05054594c5f05884db4c002c7794b139399b68
591424803611332b9eaf8517b4475dc80445f70b75b6b9530601a229be805a64529e0
74f6901788e6cfad2614d50d6c9f1f94c6c34a180b345953636b9834b9bc6b6faa569
d1fff5000b54ff7255705a71ee2925e4a3e30e41aed489a579d5595e0df13e32e1e4d
d202a7c7f68b31d6418d9845eb4d757adda6ab189e1bb340db818e5b3bc725d992faf
63e9b0500db10517fe09d3f566fba3a80e46a403e0c7d41548fbf75cf2662b00225b5
02961f98d8c9ff937de0b24c231845
group_binding_factor: f5e7fa47177f80fab386db6caa1cc4390815285748 0ddd3
7234ceba39bf741e17df4ea8d174c978ea22e1ce46bd0466b83a51be0daf3123000

// Signer round one outputs
S1 hiding_nonce: 2e5150a128bc7a396a1b0cc1f1f1fdc056a4cba6ee1abbfd7515
18cc1558209d7ba5464d4cb93e8fc1ae5a5bd2ecc479dd005ecfad19523300
S1 binding_nonce: 94745d0ad6abba6ae42d869e42a513a587a6627ab026a832f5e
f0cee24f439e0045b99795b26853f3812d839a6fdc2f9602ef74e5b39052900
S1 hiding_nonce_commitment: 473d732afb8ee9647e1ace23bf148a099c356e964
b3a22652c89008792d6a699a9db4b830b100ce16ddef6d9780d1f8b2c099dc9a40c82
7a00
S1 binding_nonce_commitment: 25a10958a534d96765609b368f35b6d36cd45db6
c5fbc00aee224671b2a37fe736dbadfb5e67161b4184d1628b378bdde07afc49a3630
0ec00
S3 hiding_nonce: ad1dfc11fd6b70c67544f63d431e7158b27157a8f794b3db03fa
4c3f06c034d115864964cc8ccff0825e17bc717ed01185a06adcfa995d3400
S3 binding_nonce: 3c4c254def4d85abd5bad3d9b50208c9c5da934c5ec14c4c004
a8e86e49f351543c14305172b6f3f429ed1c268abfbeae718aeeb90793f0200
S3 hiding_nonce_commitment: b80d36d789a1bbc65e6855067b90526cbe81d0bf3
da05054594c5f05884db4c002c7794b139399b68591424803611332b9eaf8517b4475
dc80
S3 binding_nonce_commitment: 445f70b75b6b9530601a229be805a64529e074f6
901788e6cfad2614d50d6c9f1f94c6c34a180b345953636b9834b9bc6b6faa569d1ff
f5000
```

```
// Round two parameters
participants: 1,3

// Signer round two outputs
S1 sig_share: 3df924a06903772f1c3098d98bb708bd2462011e64f184d455bea8f
5eada8a421c02538fb8fe205ac533b17870e4e3db005dddd47722e40900
S3 sig_share: 4416c13f953466f268afce361518db4b11d61639d26a36496371e11
22a1eae0a26fd863135ccc5f56799643d4039067fac11775f983bf02100

sig: c5b1a31ec5a998f190a30bf9f3023aae73cadcfbf0c7781a946b39d0f0e7ee43
90f46f11d540825e05bbd82fc6415f175f9a0302af3c131880810fe6dffe37dd2185d
f6610a1cfe30836381857365cbb1db92f8a0815f9384d42ffd9c0edcae64f2dcd15b6
b01dea5aad6e5434105ed42b00
```

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

```
// Configuration information
MAX_SIGNERS: 3
MIN_SIGNERS: 2
NUM_SIGNERS: 2

// Group input parameters
group_secret_key: 1b25a55e463cfd15cf14a5d3acc3d15053f08da49c8afcf3ab2
65f2ebc4f970b
group_public_key: e2a62f39eede11269e3bd5a7d97554f5ca384f9f6d3dd9c3c0d
05083c7254f57
message: 74657374

// Signer input parameters
S1 signer_share: 5c3430d391552f6e60ecdc093ff9f6f4488756aa6cebdbad75a7
68010b8f830e
S2 signer_share: b06fc5eac20b4f6e1b271d9df2343d843e1e1fb03c4cbb673f28
72d459ce6f01
S3 signer_share: f17e505f0e2581c6acfe54d3846a622834b5e7b50cad9a2109a9
7ba7a80d5c04

// Round one parameters
participants: 1,3
group_binding_factor_input: 00015c01341bd0a948e71fe1b9bf09f8b8ee258bf
cf3abddee42ef74c8068e0b224584a209c6c3e812283378fb6a15e4b9a64aa9eed51f
7ae405d09b56ee56bc58500003c0fc5ffaf124fa69206a9ed77bd57fa1d8ca505f613
9794f82778ce15ee0be3cb6718f8139e49d08741ab9f030da29e557451eab58bc770c
0c05ef4e2ff8001e678630bf982c566949d7f22d2aefb94f252c664216d332f34e2c8
fdcd7045f207f854504d0daa534a5b31dbdf4183be30eb4fdba4f962d8a6b69cf20c2
734043
group_binding_factor: af4288aad52765341b2238007777ea2bb2d0dfb4e92423b
0646d4bec426e3d0d

// Signer round one outputs
S1 hiding_nonce: b358743151e33d84bf00c12f71808f4103957c3e2cabab7b895c
436b5e70f90c
S1 binding_nonce: 7bd112153b9ae1ab9b31f5e78f61f5c4ca9ee67b7ea6d118179
9c409d14c350c
S1 hiding_nonce_commitment: 5c01341bd0a948e71fe1b9bf09f8b8ee258bfcf3a
bddee42ef74c8068e0b2245
S1 binding_nonce_commitment: 84a209c6c3e812283378fb6a15e4b9a64aa9eed5
1f7ae405d09b56ee56bc5850
S3 hiding_nonce: 22acad88478e0d0373a991092a322ebd1b9a2dad90451a976d0d
b3215426af0e
S3 binding_nonce: 9155e3d7bcf7cd468b980c7e20b2c77cbdfbe33a1dcae031fd8
bc6b1403f4b04
S3 hiding_nonce_commitment: c0fc5ffaf124fa69206a9ed77bd57fa1d8ca505f6
139794f82778ce15ee0be3c
S3 binding_nonce_commitment: b6718f8139e49d08741ab9f030da29e557451eab
58bc770c0c05ef4e2ff8001e

// Round two parameters
participants: 1,3

// Signer round two outputs
S1 sig_share: ff801b4e0839faa67f16dee4127b9f7fbcf5fd007900257b0e2bbc0
2cbe5e709
S3 sig_share: afdf5481023c855bf3411a5c8a5fafa92357296a078c3b80dc168f2
94cb4f504
```

sig: deae61af10e8ee48ba492573592fba547f5debeff6bd6e2024e8673584746f5e
ae6070cf0a757f027358f8409dda4e29e04c276b808c60fbea414b2c179add0e

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

```
// Configuration information
MAX_SIGNERS: 3
MIN_SIGNERS: 2
NUM_SIGNERS: 2

// Group input parameters
group_secret_key: 8ba9bba2e0fd8c4767154d35a0b7562244a4aaf6f36c8fb8735
fa48b301bd8de
group_public_key: 023a309ad94e9fe8a7ba45dfc58f38bf091959d3c99cfbd02b4
dc00585ec45ab70
message: 74657374

// Signer input parameters
S1 signer_share: 0c9c1a0fe806c184add50bbdcac913dda73e482daf95dcb9f35d
bb0d8a9f7731
S2 signer_share: 8d8e787bef0ff6c2f494ca45f4dad198c6bee01212d6c8406715
9c52e1863ad5
S3 signer_share: 0e80d6e8f6192c003b5488ce1eec8f5429587d48cf001541e713
b2d53c09d928

// Round one parameters
participants: 1,3
group_binding_factor_input: 000102688facca4e2540ef303734c5aee8e7cdba0
bc7ab94abfe63d05ebc68dde0f4c702ae64b6f0506acc3395fecd4bc70a9eb9f8e539
4264c2d2aa0c8faff857ea3058000303c21b8ee50b6478ac845c0687504db6792873f
5a327ff6a3115558070b517299302f3c73c912838f707f549bf4d63432f1fbe128fa3
5ec8ba6eca849ebd248aa46a7a753fed12531fbcd151e1d84702927c39063e780e91c
01f02bd11b60d7632bf
group_binding_factor: cf7ffe4b8ad6edb6237efaa8cbfb2dfb2fd08d163b6ad90
63720f14779a9e143

// Signer round one outputs
S1 hiding_nonce: 081617b24375e069b39f649d4c4ce2fba6e38b73e7c16759de0b
6079a22c4c7e
S1 binding_nonce: 4de5fb77d99f03a2491a83a6a4cb91ca3c82a3f34ce94cec939
174f47c9f95dd
S1 hiding_nonce_commitment: 02688facca4e2540ef303734c5aee8e7cdba0bc7a
b94abfe63d05ebc68dde0f4c7
S1 binding_nonce_commitment: 02ae64b6f0506acc3395fecd4bc70a9eb9f8e539
4264c2d2aa0c8faff857ea3058
S3 hiding_nonce: d186ea92593f83ea83181b184d41aa93493301ac2bc5b4b1767e
94d2db943e38
S3 binding_nonce: 486e2ee25a3fbc8e6399d748b077a2755fde99fa85cc24fa647
ea4ebf5811a15
S3 hiding_nonce_commitment: 03c21b8ee50b6478ac845c0687504db6792873f5a
327ff6a3115558070b5172993
S3 binding_nonce_commitment: 02f3c73c912838f707f549bf4d63432f1fbe128f
a35ec8ba6eca849ebd248aa46a

// Round two parameters
participants: 1,3

// Signer round two outputs
S1 sig_share: 9e4d8865faf8c7b3193a3b35eda3d9e12118447114b1e7d5b4809ea
28067f8a9
S3 sig_share: b7d094eab6305ae74daeed1acd31abba9ab81f638d38b72c132cb25
a5dfae1fc
```

sig: 0342c14c77f9d4ef9b8bd64fb0d7bbfdb9f8216a44e5f7bbe6ac0f3ed5e1a573
67561e1d51b129229966e92850bad5859bfee96926fad3007cd3f38639e1ffb554

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